DTIC FILE COPY

4

AD-A203 090

HOP: A Process Model for Synchronous Hardware Systems

Ganesh C. Gopalakrishnan,
Richard M. Fujimoto,

Tech. Report. UUCS-88-003

DTIC
S ELECTE D
JAN 2 6 1989
D

[illegible]
Ap[illegible]
D[illegible]

89 1 25 053

HOP: A Process Model for Synchronous Hardware Systems

by

Ganesh C. Gopalakrishnan and Richard M. Fujimoto,

Dept. of Computer Science, University of Utah,
Salt Lake City, Utah 84112, U.S.A

Phone: (801) 581-3568/8224
Email: ganesh@cs.utah.edu

UU/CS/TR-88/003

Contents

List of Figures

DTIC
COPY
INSPECTED
1

Accesion For
NTIS CRA&I ☑
DTIC TAB □
Unannounced □
Justification
By per NP
Distribution /
Availability Codes
Dist | Avail and / or Special
A-1

(was designed

1 An Introduction to HOP

A new Hardware Specification Language (HSL) called HOP is presented. HOP stands for *Hardware viewed as Objects and Processes*. It can be used for specifying the structure, behavior, and timing of digital systems.

~~We designed~~ HOP for several reasons. It integrates well-tested ideas from ~~our~~ past work ~~[GSS87,Gop86,Gop87]~~ that was based on an abstract data type ~~[GHM78]~~ view of hardware systems into a new, simple, and deterministic process model that we have invented. Our process model is inspired by the works of [Mil82], [Mil83], and [Hoa85].

it is believed

Secondly ~~we believe~~ that not only should an HSL be founded in mathematical principles, but it also ought to be simple, intuitive to use, and address practical issues, especially if practicing VLSI designers are to be encouraged to use them.

HOP was designed to meet the following design objectives:

1) Be capable of modeling large architectures as well as simple MOS digital circuits;

2) support the writing of a priori as well as a posteriori specifications;

3) possess a simple and rigorous semantics;

4) support static analysis techniques and design verification;

5) match digital designer's intuitions closely;

6) be demonstrably efficient in handling many important practical issues;

7) act as a common repository of related information falling in various domains (functional behavior, timing, geometry, and user documentation to name a few) thereby helping in designer and tool integration;

8) support design automation as well as manual design. (L'R)

In this paper we show how HOP meets objectives 2, 3, 4, 5, and 6. Objectives 1 and 7 will be addressed in the process of specifying a large Application Specific IC (ASIC) called the "Roll Back Chip" [FTG88a,Gop] that we are currently engaged in. Objective 8 will be addressed in our ongoing work on implementing a VLSI design system centered around HOP.

1.1 Features of HOP, in a Nutshell

Let us take an informal approach similar to [Mil80, Page 10] to intuitively understand HOP. The externally observable features of every hardware module modeled in HOP consist of a set of "light actuated sensors" (*input events*), "a set of lamps" (*output events*), a set of "output conduits" that *bring out data items*, and a set of "input conduits" that can *consume data items*. In addition, each module maintains "a notebook" (*the internal datapath state*) that maintains a complete record of all its input events and input data port values *in as concise a form as possible*. The notebook (internal datapath state) is visible to human observers but not to other modules. There can be modules of an extreme variety too; for instance those that have only sensors and lamps and no internal notebooks (*e.g. controllers*). The values

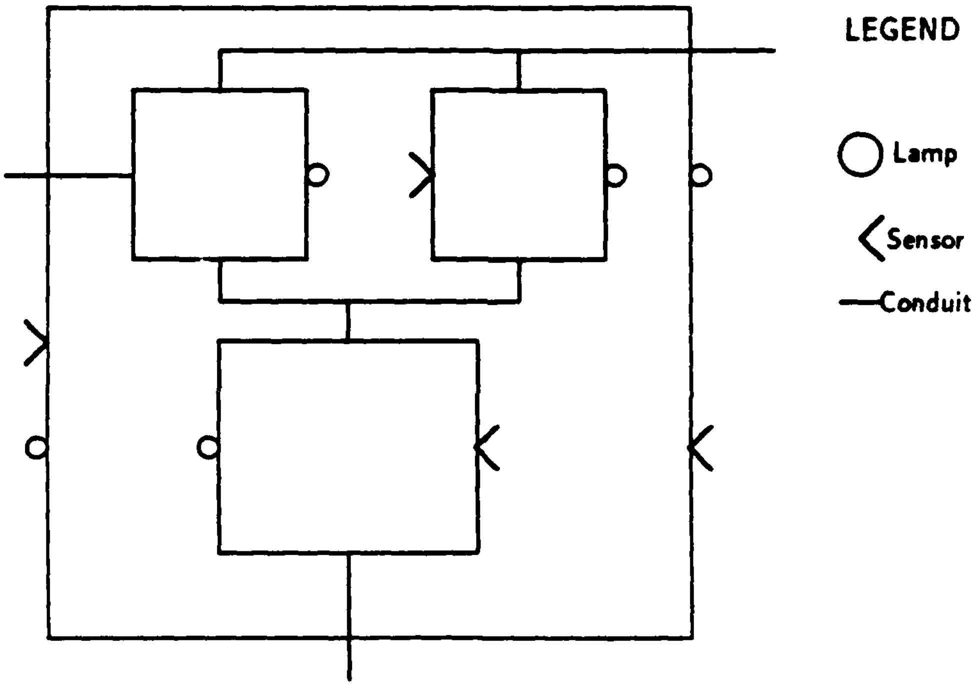

Figure 1: Informal View of HOP Processes

of data path states and the values shipped through conduits belong to one of the data types supported by HOP, such as queues, trees, bytes, or a user-defined data type.

A human observer may observe the following behavior of every module: its *current* output lamp statusses and output conduit values are entirely predictable from its *current* input sensor statusses, input conduit values and notebook contents. Further, its notebook contents at the next time step as well as its entire future behavior are also predictable. In other words HOP processes are *functions* from current input events, data values, and data path state to HOP processes. (They are deterministic.).

At each time step a module M either does or does not *wait* for one of its input sensors to be actuated (i.e. it offers a deterministic *choice* of input events). If it offers such a choice, only one input sensor may be actuated. When so actuated by another module N, *synchronization* is achieved between M and N. M may then produce and consume data values through its conduits at the current time step and makes progress in its computation. If M waits for a sensor to be actuated but none is actuated, its current actions and future computation are both undefined (*failure to synchronize*). On the other hand if M doesn't wait for any of its input sensors to be actuated, it may make progress in its computation *autonomously* after having produced and/or consumed data items at the current time step. We assume that light from lamps reach light-sensors instantaneously.

A collection of interacting modules is formed through *parallel composition*, "||". Collections of interacting modules can make progress in their computation only through synchronizations; *i.e.* every input sensor awaited by a module to be actuated should actually be actuated by some other module. All the actions through events and ports in a system of modules happen at the same rate (as in [Mil82]). If a module is busy performing internal computation, it will be regarded as outputting an output event called "idle".

Not all sensors and lamps are alike; they have different colors. A lamp and sensor may interact either if they have the same colors to begin with, or if they are imparted the same colors (this is called *renaming*). A lamp of a given color can actuate (virtually) an infinite number of sensors of that color at the time when it shines (*events have a broadcast semantics.*) Two lamps of the same color flashing at the same time is equivalent to one lamp flashing with that color.

Input and output data conduits are meant to be connected amongst themselves. A given architecture has a specific "plumbing" of the conduits. No synchronization is defined for data transfers through conduits. Thus one module may put out a value without one sampling this value, or vice versa. However modules usually achieve synchronization through their flashing lights and thereafter meaningfully interact through their conduits. Output conduits have a broadcast semantics. Two output conduits connecting to a *node* may not assert two *incompatible values* to the node (defined via a *function bus that computes the least upper bound of the values involved over a strength lattice.*). Most conduits are assigned specific directions to begin with; it is possible to have perfectly directionless conduits too.

Selected lamps, sensors, and conduits may be *hidden* from a collection of interconnected modules M_i. The collection M_i may then be viewed as a single module M that possesses only those events and ports that are not hidden. If a submodule within M waits for a sensor with color c to be actuated, no other submodule within M produces a light of color c, and if events of color c are not part of M's interface (due to hiding), then M's computation is undefined. The same goes for a lamp that shines within M without actuating any sensor within M and hidden from M's interface.

However a lamp-sensor pair that communicate within M may be hidden without any risk. The contribution of this *hidden and communicating* lamp-sensor pair to the interface of M is just an "idle" event.

Relationship to Hardware Modeling

Modules in HOP are *black-boxes* that are understood and used only in terms of their *interface*. The interface consists of *data ports*, *events*, and a *protocol specification* that uses events and asserts/queries values to/from ports.

Events are realized as different combinations of *control wires* or as predicates defined over data conduits. Module await either command events or status events. Data conduits are realized as bus structures that deliver the same data items at the receiving end as items sent at the sending end (*i.e.* the busses do not have any wire-permutations, tappings, etc.).

HOP is useful for writing both requirements (a *priori*) specifications and design (a *posteriori*) specifications. The manner in which requirements are expressed has usually no bearing on the actual implementation chosen later. Design specifications *capture known facts* about a system that has been built or has been designed in detail. In a HOP based design methodology, design proceeds hierarchically, and on many occasions (but not always) *top-down*. For most large systems, the requirements specification consists of the specification of a *collection of modules* and not *one module*; for these systems, the single module view is only derived a posteriori.

Requirements specifications are usually written without knowing at least two detailed aspects: (i) the details of the functional behavior (I/O mappings) of module(s); (ii) the details of the temporal behavior of module(s). *Abstraction mechanisms* permit modeling systems completely *despite missing details*. We employ two important abstraction mechanisms: (i) data abstractions, to model the functional aspects of the *I/O port values* as well as *data path state*; (ii) temporal abstractions in the form of a protocol description consisting of *events* and *event sequences* that describe the *control* aspects.

Most of our applications of HOP to date (as well as the examples in this paper) pertain to *synchronous hardware systems*, *i.e.* systems in which: (i) the computational rates of the modules are the same; (ii) communications between modules are lockstep synchronous with a global clock. While writing the requirements specifications for these systems however, not enough may be known about the clocking aspects. In these cases, we would pretend as if these synchronous systems were actually *asynchronous systems*—those in which all synchronizations between events happen via handshaking. Later on when a design in the synchronous style is produced, most of these "handshakes" *happen implicitly*, *i.e.* without actually exchanging any signals, but via *hard-wired assumptions built into modules*. However HOP encourages making these hard-wired timing/synchronization assumptions explicit via the introduction of *events*.

In HOP one could write a module requirements specification and later replace it by a collection of module requirements specifications. It is possible to check whether the collection is *observationally equivalent* to the (original) single module. Design specifications may also be written in HOP. Design specifications include details that closely match the details of the ultimate hardware. Thus typical design specifications of synchronous hardware systems

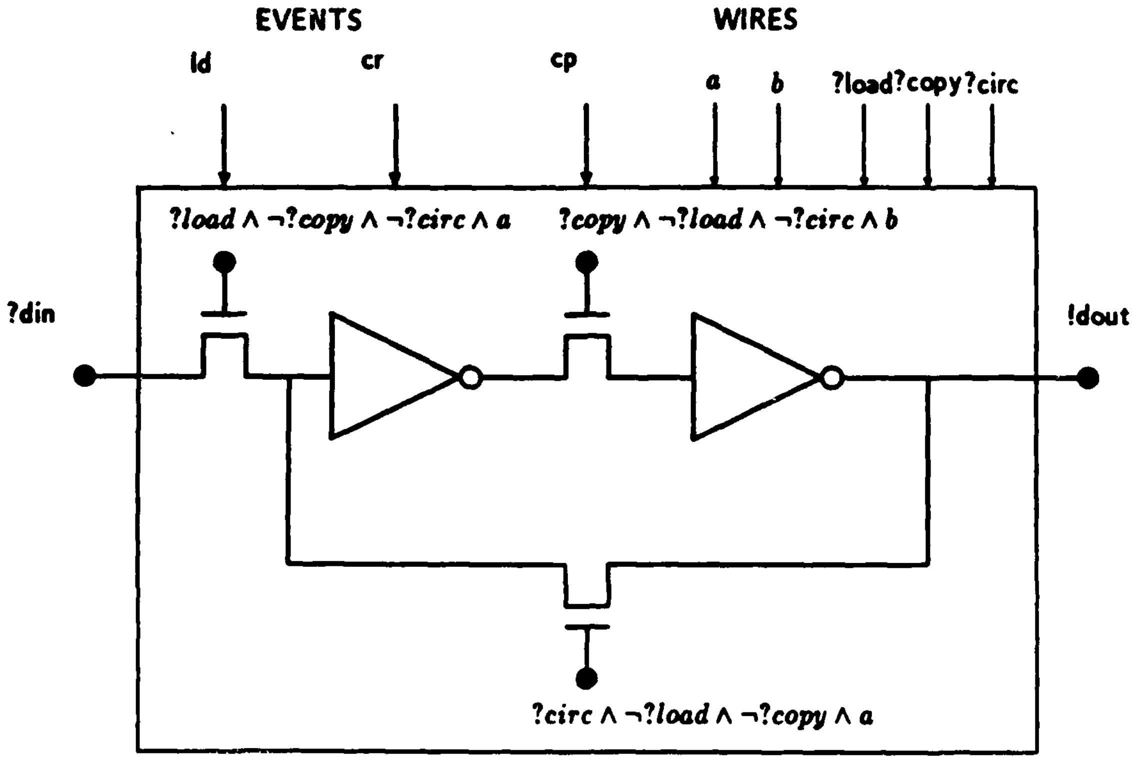
EVENTS
WIRES
ld
cr
cp
a
b
?load?copy?circ
?load ∧ ¬?copy ∧ ¬?circ ∧ a
?copy ∧ ¬?load ∧ ¬?circ ∧ b
?din
!dout
?circ ∧ ¬?load ∧ ¬?copy ∧ a

Figure 2: A master/slave Flip-Flop, FF

would include clocks.

1.2 What is Attractive About HOP?

A variety of existing as well as new ideas have been integrated into HOP. We discuss these ideas with the aid of the example in figure 2.

• *Highlighting Useful Input Combinations Thru Events:* The circuit in figure 2 may be viewed at different levels. At first glance it is an analog circuit. However it will be used as a digital system—specifically a master/slave flip-flop. Therefore not all logical combinations of the signals *load*, *copy*, and *circ* are useful. Only *three events* prove to be useful:

ld =?*load* ∧ ¬?*copy* ∧ ¬?*circ*—loads FF;
cp =?*copy* ∧ ¬?*load* ∧ ¬?*circ*—copies data to slave;
cr =?*circ* ∧ ¬?*load* ∧ ¬?*copy*—restores the charge.

HOP encourages the identification and declaration of such events. This gives rise to specifications that are easier for humans to understand and easier for programs to institute static checks on.

• *Incorporation of Multiple Clocks:* It is important to be able to incorporate multi-phase clocks and classify events based on them. Consider the usage of FF in a two-phase clocked

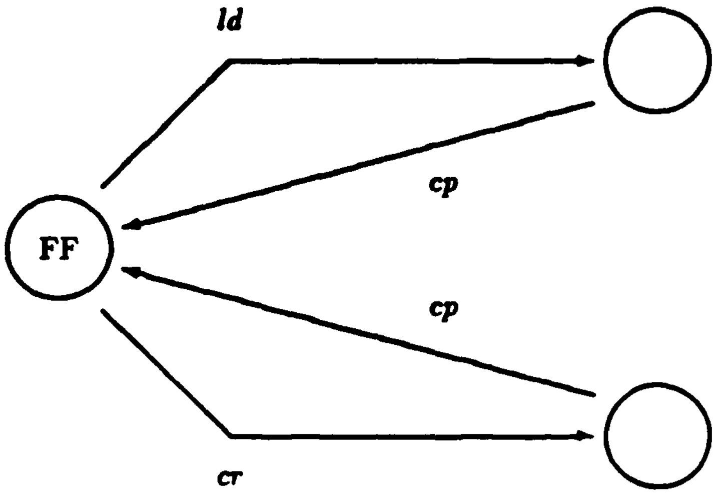

Figure 3: Excerpts from the Process Diagram of FF

system that has two non-overlapping clocks a and b. It would then be necessary to generate the above events only during specific clock phases. HOP allows this to be specified thus:

$$ld = a \wedge load \wedge \neg copy \wedge \neg circ$$

• *Highlighting Useful Event Sequences ("Modes of Behavior"):* Only certain sequences of the events ld, cp and cr are of interest. Figure 3 shows these sequences. What we have shown in this figure is actually salient excerpts from the *process diagram* of the FF. The above diagram can be expressed in the syntax of HOP as:

```
FF <=   ld -> cp -> FF
      | cr -> cp -> FF
```

• *Detection of a Class of Sequencing Errors Statically:* HOP forces designers to state the sequences of events of interest. The system can flag an error should an unspecified sequence manifest. This is easy to do due to the synchronization semantics of events. Thus for FF, a sequence ld, cp does not constitute a useful mode of behavior; were such a sequence be applied, it would be regarded as a *sequencing error*. In many traditional approaches, sequencing errors are detected in the process of simulating a circuit; to be assured of the detection of all lurking sequencing errors, a very large number of simulation test cases have

```
ABSPROC FF
 PORTS  ?din, !dout, ?ld, ?cr, ?cp : bit
 CLOCKS twophase(a,b)
 EVENTS ld = ?load /\ a /\ not(?circ) /\ not(?copy)
        cp = ?copy /\ b /\ not(?circ) /\ not(?load)
        cr = ?circ /\ a /\ not(?load) /\ not(?copy)
 PROTOCOL
  FF[dps] <=   ld, x=?din, !dout=dps   -> cp -> FF[x]
             | cr, !dout=dps           -> cp -> FF[dps]
END FF
```

Figure 4: The Complete Absproc Specification of FF

to be applied. Even then, sequencing errors are not directly noticed, but have to be deduced through *backwards reasoning* from an observed anomalous behavior, such as two values clashing on a bus. *Many sequencing errors can be detected during the process of composing two HOP processes using an algorithm called PARCOMP that we have developed.*

• *Separation of Data and Control, and Incorporation of Data I/O into Specifications:* In the design of digital systems, architects use their intuitions to separate data related aspects from control related aspects. We believe that an HSL must support this separation process. Data aspects may be loosely defined as those modes of behavior that are unaffected by the datapath states. Consider a stack as an example. For all its data path states where the stack is neither full nor empty, the same control recipe suffices. Thus, by separating data from control, we again impart a good structure to the specification.

• *Highlighting Data Related Protocols:* Continuing with the example of FF, the following important questions must be clearly answered by its specification:

- When may a user (possibly another module) read FF?
- when may the user write into FF?

Answers to such questions form the *usage protocol* of FF. Usage protocols are usually complex; for instance FF may be reliably read even while it is being loaded. We embellish the process diagram 3 to include such additional pieces of information as *annotations*. It results in figure 4 which is a complete HOP specification of FF.

This specification may be read as follows. FF is initially in control state FF and datapath state dps (a one-bit quantity). It offers the choices ld and cr to the external world. If the external world asserts ld, the input data item is also expected to be supplied at the same time through the data input port ?din. This is written as x=?din, x being the value supplied. Despite loading x, the output port !dout continues to remain at its original value which is equal to the internal data path state dps. FF then advances to a new control state (indicated by ->) where it awaits the event cp. This is generated during phase b of the two-phase clock. Thereafter FF goes back to the control state FF but in data path state x.

Similarly we may consider the path starting from FF[dps] labeled via cr and coming back to FF. In this case, FF doesn't suffer any state changes nor does it load any input values.

Although this example doesn't highlight the use of abstract data types, in general data

$$\begin{tikzcd}
\mathrm{H1}\ ,\ \mathrm{H2} \arrow[r, "\|"] \arrow[d, dash, dashed] & \mathrm{H} \arrow[d, dash, dashed] \\
\mathrm{C1}\ ,\ \mathrm{C2} \arrow[r, "\text{Connect}"] & \mathrm{C}
\end{tikzcd}$$

(Dashes indicate Equivalence

Under Zero-delay Simulation.)

Figure 5: HOP Provides a Compositional Model

path states will be modeled using high-level abstract data types (user's may introduce new abstract data types into HOP), and new data path states as well as output port values will be created using functional expressions. It is appropriate to think of HOP specifications as specifying deterministic automatons that are: (i) enriched to include information about data path states and port values; (ii) have a synchronization semantics underlying event interactions; (iii) model value communications as updates of node values over a strength lattice.

The paradigm of separation of data from control is not forced upon the designer. It may be judiciously applied when found useful. It is also possible to view control lines as data and vice versa when necessary, in a structured manner. Event to data mapping is achieved by introducing a fictitious module that awaits the event and generates a data assertion. Data to event mapping is achieved by defining one or more predicates (as needed) over the data inputs, and defining events via these predicates.

- *Compositionality:* HOP provides a *compositional model* for synchronous hardware systems as revealed by figure 5. If we have two HOP specifications H_1 and H_2 and circuits C_1 and C_2 corresponding to them, then the process of connecting C_1 and C_2 to obtain C can be paralleled in the HOP domain (essentially) by the process of applying PARCOMP to H_1 and H_2 to yield H. This property will be the basis for establishing the correctness of systems, as well as deducing behavior from structure.

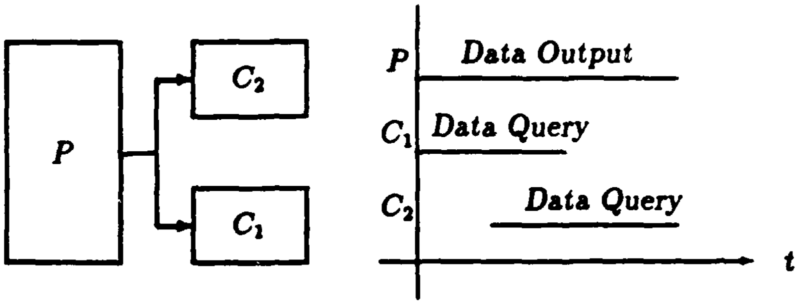

Figure 6: Illustrating Value Communication in HOP

- *Deducing Behavior from Structure:*

Suppose a collection of data path modules and controllers SM_i are connected to form a system M. Could a behavioral description for M as a black-box module be deduced automatically? That is, could we automatically obtain a behavioral description that is simple to understand because it does not require the user to visualize in his minds all possible ways in which the modules SM_i could interact?

We have developed an algorithm PARCOMP to do exactly this for HOP specifications. Numerous heuristics render PARCOMP efficient in practice.

- *Modeling Value Communication Naturally and Modularly:* A mechanism called *data actions* is used to model data transfers over data ports. This mechanism has been found to be more natural as well as *modular* to use, as opposed to synchronous value communication. This mechanism also satisfactorily models the ability for ports (busses) to perform broadcast as well as bidirectional communication. We explain this with the aid of a synchronously clocked hardware system depicted in figure 6.

In a synchronous hardware system, a module can write a data item on a bus for one or more clock ticks even in the absence of any other modules simultaneously reading from the bus. Likewise, a read can go on without any simultaneous writes. Finally there are situations such as shown in the timing diagram in figure 6. In these situations it is not appropriate to model value communication through *synchronization at every tick of the interval.* Of course one may still model these situations by forcing a synchronizing at each tick, and thereafter discarding data items "when not needed", etc. More than the awkwardness, this approach suffers from a lack of modularity of the individual specifications because the specification writer has to *anticipate* this particular context of usage of the consumer module.

Our solution involves an idea borrowed from *logical variables* as discussed in (and suggested by the author of) [Lin85]. It also relates to the work of [Bry84] and [ISD88, page 307]. We model *data assertion* as a process of imparting a *value binding* to a logical variable through a data assertion. These value bindings last only for the duration for which data assertion lasts. If no data assertion is made, the logical variable is essentially unbound. Data inputs are modeled via *data queries.* If one data assertion and several data queries are made at the same time, the queries would get the value asserted by the data assertions. Absence of queries or assertions does not cause any problems in our approach.

Multiple writers on busses are modeled as a process of imparting two value bindings to a logical variable. If these values do not agree, the binding associated with the logical variable is *error*. *Agreement* is defined through a *bus* function that implements a monotonic mapping over a *strength lattice* (similar to [Bry84]). Such a strength lattice is defined for every type of value that can be communicated over ports. For the *bit type* (extension of the *boolean type*) the lattice includes the *strong values* 0,1,U, and bit-error, where U stands for 0 or 1, but Unknown; *e.g.* the state of an un-initialized flip-flop is a U *bit*. The only *weak value*—one that can be dominated by the strong values—is Z which stands for high-impedance. We model transistor switches as devices that generate a Z value when open. Thus when module M_1 drives a bus through an open switch and M_2 puts a 1 on the bus through a closed switch, the net state of the bus would be determined by $bus(Z, 1)$ which is 1. Thus the bus would be bound to 1. In HOP, bidirectional switches are modeled as devices that force agreement between two logical variables.

A good way to look at value communication in HOP is that *the proper synchronization of events guide data queries and assertions into a correct, implicit synchronization.*

- *Modeling "Arhythmic Arrays"* A majority of examples published in the area of specification driven design are architectures consisting of dissimilar modules [Mos83,CGM86,Coh88, Hun87]. Regular arrays have received comparatively lesser attention ([She84,She85], [Pat85], [MH85], [BW]). Among regular arrays, most examples have involved pipelined or systolic arrays. Most geometrically regular arrays are however *not* computationally regular. We call such arrays *arhythmic arrays*.[1]Systolic arrays are a special case of arhythmic arrays where both the geometry and the computations are regular. Some examples of arhythmic arrays are registers, random-access and content addressable memories, FIFO queues, shift registers, various types of carry-chains, and the LRU matrix discussed in section A.

Issues in the specification and verification of arhythmic arrays are different from those for systolic arrays. Systolic systems typically effect data transformations on streams of data, each member of the array essentially invoking the same operation on the elements of the stream. Members of arhythmic arrays support multiple modes of activity. During a given time interval, different members of an arhythmic array are involved in different modes of activity. Also in arhythmic arrays, geometrical issues are closely coupled with behavioral issues.

HOP addresses both behavioral and geometric issues quite effectively. We have developed an efficient *divide and conquer* technique for performing PARCOMP on arhythmic arrays. We believe that HOP could be effectively used for specifying systolic systems by following the approaches taken by [She85] or [Hen84].

- *Abstraction Mechanisms:* In addition to behavioral and temporal abstractions of HOP discussed in section 1.1, HOP supports data and structural abstractions also[2]. Structural abstraction is achieved by the process of selectively hiding internal connections among ports and among events. Behavioral abstraction is achieved by the introduction of processes and mathematical functions that *model* the actual behavior. Data abstraction is the use of a variety of user-defined data types to model state and port values. Two varieties of data types are supported in HOP: (i) equationally defined abstract data types, similar to [GHM78]; (ii) data types defined via abstract models, similar to [LS75].

There are two approaches to specifying external timing *requirements*: (i) specify the most

[1] In the same "hearty spirit" as the word "systolic"!

[2] A discussion of these four abstraction mechanisms appears in [BP88, Chapter 9].

general temporal behavior admissible; (ii) specify concrete bounds on the timing of various modes of activity. We follow the former approach in this paper.

In order to amplify this point, consider synchronous hardware systems as an example. In writing the requirements specification of synchronous systems in HOP we *actually pretend that they behave similar to self-timed hardware with handshaking events*. These handshaking events are merely conceptual in nature. When the actual system gets designed, the designer gives definitions for these conceptual events. For example for the *push* operation on a stack, we will associate a *davail* event to "notify" the stack when the data to be pushed on it is available. In the actual implementation, we will not have this handshake line, but instead a *discipline* for using the *push* operation, such as: "the user is expected to supply the data exactly one tick after applying *push*." (This example also points out that it seems attractive to define HOP's events in temporal logic.)

This approach to timing has two advantages:

- Implicit (hard-wired) timing assumptions in synchronous hardware are made explicit; We believe that hard-wired synchronization assumptions are even worse than hard-wired constants in programs, and are a source of common sequencing errors. Synchronous hardware designs where synchronization is hard-wired are difficult to modify and reuse.
- If the conceptual events are actually implemented as signals, we get a self-timed implementation of the system. Thus a common specification serves both synchronous and asynchronous implementations.

1.3 Comparison With Related Research

We compare HOP with Circal[Mil85a,Mil83], [Gor81], Johnson[Joh84], HOL[CGM86], and SBL[GSS87], using FF as an example.

1.3.1 Circal

The examples reported to date suggest that Circal attempts to model systems with considerably more detail than we care to do in HOP. For instance, FF would be modeled by modeling wires, inverters, and pass transistors as having some propagation delay. Though published examples in Circal have not emphasized the identification of useful *events* and modes of behavior, in principle this is possible to do.

Circal and HOP share the common feature of taking a process oriented view. However a crucial difference exists in the way value communication is performed. In Circal, data communication between modules over a port that can carry data items of type T is modeled as synchronous communication over a sort of *labels* l_i where i ranges over the value sort of T. In our experience, HOP's approach is more convenient to use for large architectures that are specified at the system clocking level.

1.3.2 Next-state and Output Function Based Approaches

[Gor81] and Johnson[Joh84] correspond to a modeling style where the next state and current outputs are functionally determined by the current state and current inputs. HOP's process

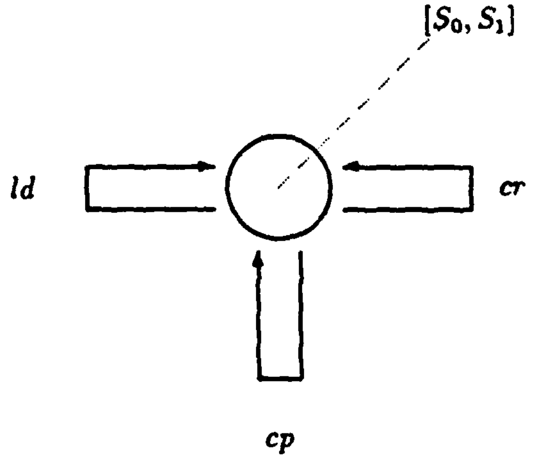

Figure 7: An Undesirable Process Diagram for FF

diagrams can be made to correspond to this model simply by using only one control state always, and modeling the rest of the state of the system as data path states. A process diagram for FF corresponding to this view is shown in figure 7. In this approach, we explicitly model both the bits stored inside the FF. We then define next data path states for the inputs 1d, cp, and cr.

However notice that this diagram does not prevent the sequence '1d, cr' from being applied. This is *not* a useful mode of use of FF. Besides this approach is of lower level because it requires both the storage nodes to be made explicit whereas ideally the data path state must only be an *abstract model* of the state of the system. We have noticed both these problems in the approaches taken by [Gor81] and Johnson[Joh84].

Finally, the next-state and output function based approaches do not support the notion of 'synchronization failure'. We believe that the static checks instituted by HOP based on event sequences is a form of "temporal type checking" that is promising in the early detection of sequencing errors. It forces designers to state their sequencing assumptions and supports the checking of these assumptions for them.

1.3.3 HOL

The style in which HOL specifications have been presented in publications so far does not match hardware designer's intuitions very well in one regard: instead of talking about the internal states of modules, HOL introduces a higher order relation to model relations between port signals.

We believe that internal states are a very intuitive "reality" in hardware. Besides, states are nothing but *equivalent classes* of I/O histories; thus there is an inherent notational economy in a state based representation.

In contrast to HOP, the style of temporal abstraction followed in HOL [CGM86, Multiplier Example] is to introduce an existential quantifier that says: "there exists a future time where the action in question happens". This approach is less operational (hence less intuitive for practicing hardware designers). It also does not introduce events that correspond to points in time where some crucial interactions between modules take place. The introduction of such events in HOP makes specifications more readable and more amenable to static analysis. It also supports self-timed implementations directly.

1.3.4 SBL

HOP evolved out of SBL [GSS87,Gop86]. SBL modeled hardware systems as abstract data types with a set of external operations corresponding to state changing (*constructor*) and port-value producing (*observer*) operations. These operations have associated timing characteristics. HOP was created to overcome certain restrictions in SBL's ability to model complex timings. Also HOP treats controllers as well as data path elements without distinction as processes; SBL was organized based on a centralized controller discipline.

As in SBL, in HOP internal states of modules as well as values communicated over ports are modeled using abstract data types, A fairly rich type system exists in HOP. HOP combines the best of process based models and abstract data type based models.

The purely algebraic approach based on SBL is still being pursued by the second and third authors of [GSS87], and SBL has independently matured considerably since the time

HOP was created.

1.4 Organization of the Paper

Section 2 presents our design methodology and the highlights of the language. The operational semantics of HOP is also presented here. Section 3 presents the specification of two versions of a simple stack. Section 4 considers the formal verification of these two versions of the stack. Section 5 presents two versions of the PARCOMP algorithm. The current implementation as well as future directions of research are presented in section 6.

2 Terminology and Operational Semantics

2.1 Terminology

There are three kinds of HOP specifications: *absproc*, *realproc*, and *vecproc*.

An absproc specifies a module as a black-box. It specifies the *interface* of the module, consisting of *data ports*, a set of *events*, and a *protocol* specification. A vecproc (a special case of realproc) is tailored for specifying arhythmic arrays.

A realproc specifies the realization of a module as a heterogeneous collection of submodules. In this paper we do not consider the process of picking the realization that best suits the problem in hand; we only address modeling of the selected realization. A realproc is a three-tuple $< S_i, C, E >$ where S_i is a collection of module specifications (absproc or realproc), C is an *interconnection*, and E is an *export-list*.

An interconnection is the union of *data interconnections* and *event interconnections*. A data interconnection is a binary relation over data ports, and indicates the ports that are connected. An event interconnection is a binary relation over events, and indicates those events that are forced to occur at the same time (by tying control wires together, for example). An export-list is the union of data export list and event export list. Data export lists and event export lists are subsets of data ports and events (respectively), and indicate those ports and events of the submodules that are part of the interface of the realproc.

Primitive modules are modules whose design refinement in HOP is of no interest. Thus only an absproc of primitive modules is of interest. For every primitive module M, a behaviorally identical circuit C is assumed to be available.

The HOP design methodology for designing a module M takes one of the following approaches (recursively defined):

Top-down1 An absproc specification for M (M_{ap}) is written, followed by a realproc M_{rp}. PARCOMP is used to infer an absproc specification for M. The inferred absproc for M is called M_{api}. The behavior observable at the interface of M_{ap} and M_{api} are then compared for agreement. At present this is supported by a manual verification methodology. If an agreement exists, the designer then proceeds to apply the HOP design methodology to the submodules of M.

Top-down2 The designer does not write M_{ap}, but begins by writing M_{rp}. M_{api} is inferred using PARCOMP, and then M_{api} is studied either manually, through simulation, or

$$
\begin{aligned}
e,\, e &\Rightarrow e \qquad (1)\\
e,\, \bar{e} &\Rightarrow \bar{\bar{e}} \qquad (2)\\
e,\, \bar{\bar{e}} &\Rightarrow \bar{\bar{e}} \qquad (3)\\
\bar{e},\, \bar{e} &\Rightarrow \bar{e} \qquad (4)\\
\bar{\bar{e}},\, \bar{e} &\Rightarrow \bar{\bar{e}} \qquad (5)\\
\bar{\bar{e}},\, \bar{\bar{e}} &\Rightarrow \bar{\bar{e}} \qquad (6)\\
!p = E_1,\ !p = E_2 &\Rightarrow\ !p = bus(E_1, E_2) \qquad (7)
\end{aligned}
$$

Figure 8: Definition of *Action Product* in HOP

through formal verification to confirm that it has the desired behavior. The HOP design methodology is then applied to the submodules of M.

Bottom-up A *partial* M_{rp} is written; specifically, its submodules are selected but the connections are not determined. The HOP design methodology is then applied to these submodules. M_{rp} is completed by providing the interconnection list and the export-list. M_{api} is then inferred from M_{rp} using PARCOMP, and is then examined for correctness.

2.2 An Operational Semantics for HOP

This section is organized as follows. We begin by defining *action*, the basic unit of communication activity that a process may engage in. Actions are either *events* or *data actions*. Both events and data actions are further sub-classified. The domain of actions for a process is *act*. A set of simultaneous actions is known as a *compound action*.

We then define reduction rules for compound actions. These reduction rules are based on the notion of *action product*, as in [Mil82]. Our action product operator is the infix operator ','. We then define a process as a system that engages in a compound action *ca* at the current time and transforms itself into a new process that begins its activity at the following time step.

Thus the meaning of a HOP process is its *transition relation* $\xrightarrow{ca} = Proc \times act \times Proc$ which is defined via structural induction over the *abstract syntax* of HOP. The definition of $\xrightarrow{ca}$ is the operational semantics of HOP. We will define new HOP processes from existing ones by using the notation $\frac{ante}{conse}$ where *ante* is an already defined HOP process (the "antecedent"), and *conse* (the "consequent") introduces the next syntactic category of processes that has not been defined so far.

2.2.1 Actions, and Action Product

Events in HOP consist of *input events* written as e, *output events* written as $\bar{e}$, and *synchronized events* written as $\bar{\bar{e}}$.

An input event e represents a logical condition that is awaited (at some time) by a module. An output event $\bar{e}$ represents the satisfaction of a logical condition at a particular time instant. The notion of synchronized events $\overline{\overline{e}}$ was introduced in HOP to impart a *broadcast* semantics to output events. Let us examine synchronized events in detail.

A synchronized event $\overline{\overline{e}}$ represents three facts: (i) at the time $\overline{\overline{e}}$ is generated, an output event $\overline{e}$ has synchronized with one or more input events e; (ii) because an $\overline{e}$ has a broadcast semantics, $\overline{\overline{e}}$ also has a broadcast semantics; therefore $\overline{e}$ as well as $\overline{\overline{e}}$ look the same as far as an input event e is considered; (iii) $\overline{\overline{e}}$ and $\overline{e}$ are treated *differently* by the 'hiding' operator of HOP: hiding $\overline{\overline{e}}$ results in an $\overline{idle}$ transition (similar to the τ of [Mil82]); however hiding $\overline{e}$ causes the synchronization tree to be pruned. This is because $\overline{\overline{e}}$ represents a mode of behavior that will be selected because of synchronizations, whereas $\overline{e}$ represents a mode of behavior that will not be selected because it *has not synchronized so far*, and *it is going to be hidden*. We find the usage of $\overline{\overline{e}}$ to be more convenient than the mechanism of γ-conjunction proposed in [Mil82, page 32] to model broadcast.

Data actions have only one simplification rule defined for them by action product: when two different data assertions $!p = E_1$ and $!p = E_2$ are made, the resultant value on the port $!p$ is defined by the function $bus(E_1, E_2)$. A complete definition of the action product operator is given in figure 8.

2.2.2 Definition of the Transition Relation $\xrightarrow{ca}$

In this section, we define the transition relation by structural induction. Before these definitions are applied to a realproc or a vecproc, all the port and event names in their submodules are assumed to be renamed so as to be distinct. Also, every compound action used in a definition is assumed to be irreducible under the action product operator ','.

Process STOP

STOP is the simplest of HOP processes. It has a null transition relation; *i.e.* it always remains halted.

A *finite process* is defined to be one that will become STOP in a finite number of steps. A finite process does not usually represent any practically useful hardware system. Therefore if PARCOMP results in a finite process starting from non-finite processes, there is room for suspicion that there are synchronization errors in the system. This is how we detect sequencing errors statically during PARCOMP: the reason for giving rise to a finite process can usually be pinpointed as a *collection of unsynchronized events that are hidden.*

Sequential Processes

$\boxed{\textit{Action:}\quad (ca \rightarrow P) \xrightarrow{ca} P}$

If P is a process, $ca \rightarrow P$ is a process that first performs the compound action ca and then behaves like P. Since actions are performed through *mutual cooperation*, the correct way to look at the process $P = e \rightarrow P'$ is that P has the potential to perform e and continue to behave line P'. If ca involves no events at all, the process can always make progress.

Vacuous compound actions are flagged by a single output event $\overline{idle}$. Thus a process $\overline{idle} \rightarrow P$ performs an idling step and continues to behave like P. $\overline{\epsilon}$ is an identity element of the action product operator ','. Most commonly, $\overline{idle}$ is introduced in a specification as a result of hiding a synchronized event $\overline{\overline{e}}$.

Sequential Processes are a special case of *deterministic choices* where there is exactly one choice available.

Deterministic Choice

$$\boxed{\textit{Det-choice: } (|_i\, ca_i \rightarrow P_i) \xrightarrow{ca_i} P_i}$$

The next category of HOP processes considered is the *deterministic choice*. A process $P = |_i\, ca_i \rightarrow P_i$, where i ranges over an index set I is one that offers a *deterministic choice* consisting of the compound actions ca_i during its first computational step. If choice c_M is accepted, P continues to behave like P_M. Example: The *FF* module of section 1 offers a deterministic choice of the events *ld* and *cr* during the first time step.

If I has more than one element, then:

1. *There must be an input event e_i present in each ca_i. Since the e_is govern the selection of one of the alternatives of the choices, the e_is must be pairwise mutually exclusive.* Since input events are boolean expressions, two events e_i and e_j are mutually exclusive if their conjunction is equivalent to *false*. This fact is almost always decidable in practice because events are usually defined as boolean expressions. However, HOP does allow events of a more general nature to be defined, using user-defined predicates belonging to a Turing-complete language. In such cases, well-formedness checks for mutual exclusion of events cannot always be carried out. In practice this situation is not expected to arise frequently.

2. *Data queries may appear in an unrestricted manner among the ca_i and they do not govern the choice.* This is only to enfore a discipline on the use of the choice construct. It is still possible to implement choices based on current "data inputs", by defining events that correspond to these data inputs—such as $?port = 55$.

A deterministic choice process, such as P (above), can be depicted as a tree where the root node of the tree corresponds to P, and there are arcs labeled with ca_i leading from the root nodes of P to the root nodes of P_i. Process diagrams are a finite representation of these trees. These trees are structurally similar to the synchronization trees of [Mil82]. However, note again the absence of nondeterminism in HOP.

Adding Actions To Initials

If P is a process, $ca1, P$ is a process which adds $ca1$ to the initials of P. Further, $ca1, P$ must obey the restrictions defined for deterministic choices (mutually exclusive guards. and the same data assertions in all the branches):

$$\boxed{\textit{Add-to-initials: } \frac{P \xrightarrow{ca} P'}{ca1, P \xrightarrow{ca1,ca} P'}}$$

Hiding

"Hiding an event e" is a shorthand for saying that e, $\overline{e}$, and $\overline{\overline{e}}$ are all hidden from a process. In the rule *Hiding-sync*, we are considering the hiding of $\overline{\overline{e}}$. Since $\overline{\overline{e}}$ represents an event resulting from a synchronization, hiding $\overline{e}$ is considered safe; we merely replace $\overline{e}$ by $\overline{idle}$. This model: the ability to drive several control inputs from one source and internalizing the connections:

$$\boxed{\textit{Hiding-sync}\quad \frac{P \xrightarrow{ca} P'}{\texttt{Hide}\ e\ \texttt{in}\ P \xrightarrow{ca[\overline{idle}/\overline{e}]} \texttt{Hide}\ e\ \texttt{in}\ P'}}$$

The notation "[new/old]" is used to mean that "new" replaces "old".

Hiding e or $\overline{e}$ from a process prevents it from synchronizing on these symbols. This can be captured by *pruning* those branches of the synchronization tree that are labeled by e or $\overline{e}$:

$$\boxed{\textit{Hiding-unsync}\quad \frac{P \xrightarrow{ca1} P',\ P \xrightarrow{ca2} P'',\ e\ or\ \overline{e} \in ca1}{(\texttt{Hide}\ e\ \texttt{in}\ P) \xrightarrow{ca2} (\texttt{Hide}\ e\ \texttt{in} P'')}}$$

Hiding a data output port removes data assertions made on that port from the current compound-action of the process. This would affect those processes that perform a data query from a connected port at the same time:

$$\boxed{\textit{Hiding-dout}\quad \frac{P \xrightarrow{ca,!p=E} P'}{\texttt{Hide}\ p\ \texttt{in}\ P \xrightarrow{ca} \texttt{Hide}\ p\ \texttt{in}\ P'}}$$

Hiding a data input port causes those variables that would have been bound by a data query on this port to remain unbound:

$$\boxed{\textit{Hiding-din}\quad \frac{P \xrightarrow{ca,x=?p} P'}{\texttt{Hide}\ p\ \texttt{in}\ P \xrightarrow{ca} \texttt{Hide}\ p\ \texttt{in}\ P' with\ x\ free\ in\ P'}}$$

Renaming

Processes are made to interact with each other either via events or via data actions on ports, by renaming those events and ports to common names:[3]

$$\boxed{\textit{Renaming-}\epsilon\quad \frac{P \xrightarrow{e} P'}{\texttt{Rename}\ e\ \texttt{to}\ e1\ \texttt{in}\ P \xrightarrow{e1} \texttt{Rename}\ e\ \texttt{to}\ e1\ \texttt{in} P'}}$$

$$\boxed{\textit{Renaming-}\overline{e}\quad \frac{P \xrightarrow{\overline{e}} P'}{\texttt{Rename}\ e\ \texttt{to}\ e1\ \texttt{in}\ P \xrightarrow{\overline{e1}} \texttt{Rename}\ e\ \texttt{to}\ e1\ \texttt{in} P'}}$$

$$\boxed{\textit{Renaming-port}\quad \frac{P \xrightarrow{da} P',\ da\ uses\ p}{\texttt{Rename}\ p\ \texttt{to}\ p1\ \texttt{in}\ P \xrightarrow{da[p1/p]} \texttt{Rename}\ p\ \texttt{to}\ p1\ \texttt{in} P'}}$$

Parallel Composition

The parallel composition operator || models the process of realizing a system by putting together several sub-processes, and permitting their interaction through events and ports that are connected. In HOP, ports having the same name *sans* the ? and ! symbols are connected. Likewise, events having the same name *sans* the overbar are connected:

[3] Renaming events as well as ports implies the appropriate use of connections as well as "glue logic" in the underlying hardware

$$\boxed{\textit{Parcomp}\ \frac{P \xrightarrow{ca1} P',\ Q \xrightarrow{ca2} Q'}{(P\|Q) \xrightarrow{ca1,ca2} (P'\|Q')}}$$

In the above definition, we assume that the mutually exclusive nature of the choices offered by one process is not disrupted by the other process. E.g. if P offers e_1 and e_2, then we assume that Q does not generate '$\overline{e_1}, \overline{e_2}$'. A syntactic definition of the *Parcomp* rule that strengthens the precondition to this effect is more involved and hence not shown here.

After performing parallel composition according to the above rule, we may simplify the result by using the following rule (if applicable). This rule captures the effect of value communication:

$$\boxed{\textit{Value Communication During Parallel Composition}\ \frac{P \xrightarrow{(x=?p),(!p=E),ca} P'}{P \xrightarrow{(!p=E),ca} P'\ [E/x]}}$$

Conditionals

HOP processes are usually defined as process schemas $P[dps]$, where for each value of dps we have one specific process. dps usually represents the data path state of the process. We have the notion of *conditional processes* in HOP that allows us to specify the behavior of a process based on its dps variable. Thus we may define a process P as:

$$P[dps] \Leftarrow if\ p(dps)\ then\ P1[f(dps)]\ else\ P2[g(dps)].$$

After reducing the predicate application $p(dps)$ to *true* or *false*, one of the following rules would apply:

$$\boxed{\textit{Conditional}\ \frac{P1 \xrightarrow{ca} P'}{(\texttt{if}\ true\ \texttt{then}\ P1\ \texttt{else}\ P2) \xrightarrow{ca} P'}\ ;\ \frac{P2 \xrightarrow{ca} P'}{(\texttt{if}\ false\ \texttt{then}\ P1\ \texttt{else}\ P2) \xrightarrow{ca} P'}}$$

Recursion

A collection of one or more processes may be defined recursively. The following rule (adapted from, and explained in [Mil82]) applies:

$$\boxed{\textit{Recursion}\ \frac{P_i[fix\ \overline{X}.\overline{P}/\overline{X}] \xrightarrow{ca} P'}{fix_i\ \overline{X}.\overline{P} \xrightarrow{ca} P'}}$$

In this paper, it suffices to view recursion as iteration.

Indefinite Delay

T' e phrase "$\rightsquigarrow$ e" stands for: "Delay indefinitely until e occurs." Its definition is as follows:

$$\begin{array}{lcl} P1 & \Leftarrow & ca1 \rightsquigarrow e, ca2 \rightarrow R1 \\ & \textit{is equivalent to} & \\ P1 & \Leftarrow & ca1 \rightarrow Q1 \\ Q1 & \Leftarrow & not(e) \rightarrow Q1 \\ & & \mid e, ca2 \rightarrow R1 \end{array}$$

2.3 An Assessment of the Merits of the HOP Model

Although a formal definition of the underlying semantic model of a language is always desirable, the practical utility of the model has to be separatively established. We now offer our own supportive comments as well as existence proofs for the merits of HOP's model. Our approach is partly motivated by that taken in [Mil82].

2.3.1 From Intuitions to Operational Semantics

Since an operational semantics is a compact embodiment of intuitions, it is prone to disagreement. Consider for instance the definition of action product. It may be argued that it is acceptable to consider $\bar{c}, \bar{c}$ as both $\bar{c}$ as well as *error* based on whether driving a control wire with a 1 from two sources is normal or an error (due to the danger of skews, for example). Our decision in this regard is to treat $\bar{c}, \bar{c}$ as $\bar{c}$, but issue a warning in the actual implementation.

Yet another place where the HOP model can fail its users if used improperly is related to *set-up* and *hold* times. Consider a read/write memory. Throughout a write cycle of the memory, the address input must remain stable, lest an unintended location get written into. The data input is allowed to change within the write cycle so long as it stabilizes a fixed duration before the end of the write cycle. The idealized timing model taken in HOP (or for that matter in [Gor81], [CGM86], and [Joh84]) does not make a distinction between these temporal types. Hence it is possible to prove a design to be correct without implying the correctness of the corresponding circuit.

Our solution is as follows. We borrow ideas from past researchers who have attempted to classify signals into different *temporal types*, such as T (*stable throughout*), E (*stable towards the end*), etc. [Noi82,Kar84]. We would take a staged approach where a HOP verification would be followed by a circuit-theoretic reasoning based on temporal types. This would provide more reliable validation in addition to partitioning concerns (verification in an idealized model is separated from checking for proper set-up/hold times).

2.3.2 From An Operational Semantics to a Denotational Semantics

Just as *trace sets* are denotations of deterministic CSP processes [Hoa85, Chapter-2], *Trace-Nodebinding* sets (TN sets) are the denotation of HOP processes. Traces have the same meaning as in [Hoa85], and nodebindings capture the effect of value assertions on nodes.

If P is a process, we define a ***meaning function*** $\mathcal{M}$ such that

$$\mathcal{M}(P) = \{< t_1, \sigma_1 >, < t_2, \sigma_2 >, \ldots\}.$$

A TN pair $< t_i, \sigma_i >$ is in $\mathcal{M}(\mathcal{P})$ if and only if process P can perform a sequence of compound events t_i while generating a sequence of node value bindings, σ_i. Both t_i and σ_i are prefix-closed sets. The TN set is obtainable from the transition relation $\rightarrow$.

Alternatively, it is possible to assign a Kahn semantics [Kah74] to HOP. Events will then be regarded as bit-streams, and data I/O as general streams. A state-stream that feeds back into the module will capture the updating of the data path state.

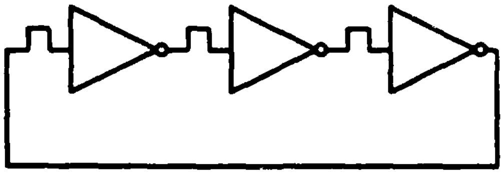

Figure 9: Either a Shift Register or a Ring Oscillator!

2.3.3 Identification of Equational Laws

Due to the absence of nondeterminism in HOP, its equational laws are simple in nature (such as: the action product operator ',' is commutative and associative; the || operator is commutative and associative, etc.).

Even in the absence of nondeterminism, we have identified several useful and non-trivial notions of *equivalence* between HOP processes. For example, the following questions arise during the process of verification and optimization (say, via *pipelining*):

- how do we relate requirements specifications to design specifications in the process of verification?
- in what sense is a pipelined system comparable to its non-pipelined counterpart?

In both the above cases, the identity relation between TN sets is far too strong to be useful. We address these questions in section 4.

2.3.4 Handling Examples That Go Beyond the Limits of the Model

Consider the circuit in figure 9. If we admit an event corresponding to the closing of all the three switches, the node values attained in this circuit would be equal to the solution of the equation $x = not(x)$, *i.e.* the undefined element BIT-ERROR. All operators are *strict* on BIT-ERROR, as BIT-ERROR is the *top* element of the BIT value lattice.

In short:

- we cannot rule out any circuits based on their *structure*;
- though circuits could become unusable with respect to those ports that generate BIT-ERROR, they could well be used with respect to other ports.

2.3.5 Relationship to Automata-theoretic Models

HOP specifications that put a finite bound on the data path state and port value types' value sorts can be regarded as a collection of communicating *Mealy Machines* [HU79]; a process

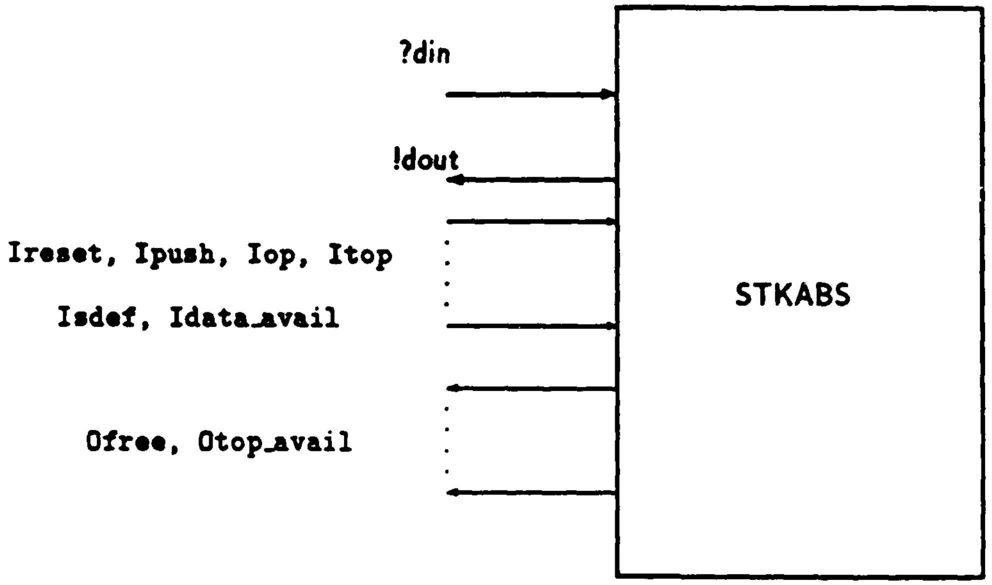

Figure 10: Schematic of the absproc of a Stack

turning into STOP will then correspond to an error-transition made by an automaton. In HOP, we prevent the state explosion that can result in modeling systems such as memory arrays using Mealy machines, by factoring system states into *data* and *control* states. For a memory module, the number of control states are independent of the memory size. The number of data path states do not concern us as data path states are modeled using abstract data types, and data path state updates and observations are modeled in a first-order functional programming language[Hen80].

2.3.6 Supportive Examples

Our primary source of confidence in HOP derives from the success we have had in specifying a wide spectrum of digital systems, both small and large as well as complex. Due to the shortage of space, we examine only relatively simple examples in this paper.

3 Specifications of a Stack

3.1 Absproc of an Unbounded Stack

```
stkabs [dps] <=
   Ireset ~> Ofree -> stkabs[reset(dps)]
 | Ipush  ~> Idata_avail, vdin = ?din ~> Ofree -> stkabs[push(dps,vdin)]
 | Ipop   ~> Ofree -> stkabs[pop(dps)]
 | Itop   ~> Otop_avail, !dout=top(dps) ~> Ofree -> stkabs[dps]
 | Isdef  ~> Ofree -> stkabs[dps]
```

Figure 11: Requirements Specification for a Stack

As defined in section 2, the *interface* of a process consists of a set of data ports, a set of events and a protocol specification. Figures 10 and 11 specify the interface of an *unbounded stack*, **stkabs**. The stack operates as follows. It supports the operations *reset*, *push*, *pop*, *top*, and *sdef* (similar to "no op"). The first three modify the state of the stack in an obvious way. After a *top* operation, the current top of stack is made available on the output port *!dout*. *sdef* corresponds to a "no op".

We will now write a *requirements specification* for the stack. In this specification, we model the stack's data-path state via the *stack abstract data type*. We specify the timing of the stack in a manner that is uncommitted to any particular clocking discipline. This requires that we adopt a few conventions so that when a design specification of the stack is available, it becomes possible to relate it to the requirements specification:

- *Generate an output event along with every data output.* This event *announces* the availability of the data output. For a set of simultaneous data assertions, we need introduce only one such output event.
- *Introduce an input event along with every set of simultaneous data queries.* This input event signifies data availability.
- *Do not insist on any specific delays between events.*
- *Notify the completion of an "operation" by a "free" event.*

The specification in figure 11 follows these conventions. For ease of typing, we denote an input event e using `Ie`, an output event $\bar{e}$ using `Oe`, and a synchronized event $\bar{\bar{e}}$ using `Se`.

Let us examine the *push* operation. It is selected by applying the corresponding output event `Opush` that matches the event `Ipush` offered by the stack. **stkabs** then waits indefinitely for event `Idata_avail`. When `Idata_avail` is asserted by a module M, a module N (possibly the same as M) is expected to bind the port `?din` with the value to be pushed. Therefore `vdin` gets bound to this value. Thereafter, the **stkabs** process performs internal activities that last an unspecified amount of time. These activities stop when event `Ofree` occurs. One step after `Ofree` occurs, **stkabs** goes back to its top-level control state, ready to accept the next command from the external world. In going to the top-level control state, the data path state is changed to `push(dps,vdin)`. This is signified by the expression **stkabs**`[push(dps,vdin)]`.

Now consider the *top* operation. Once triggered, **stkabs** goes into a period of internal activity that is terminated by the occurrence of the event `Otop_avail`. When this happens the value binding on the output port `!dout` is the "top of stack", `top(dps)`. After some more unspecified delay and one step after event `Ofree` occurs, the stack returns to its top-level control state to accept its next command.

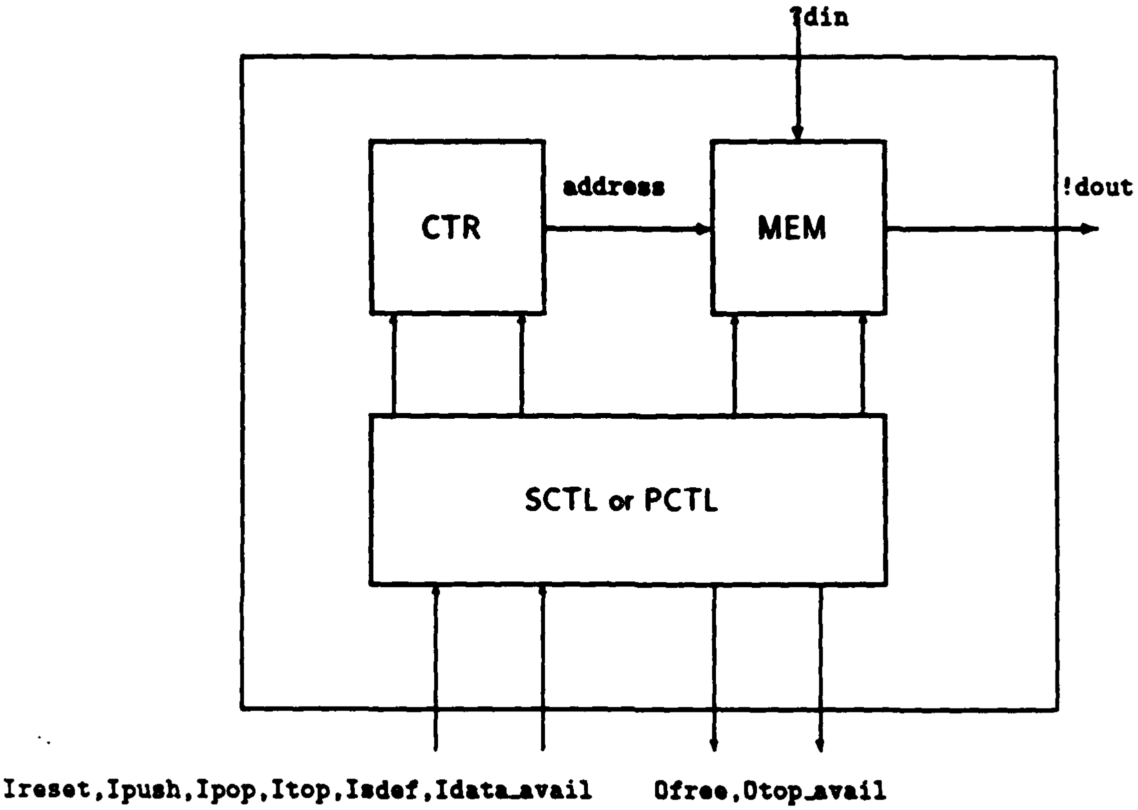

Figure 12: Schematic of the Realproc of a Stack

3.2 Realproc of the Unbounded Stack

The schematic in figure 12 is intended to implement the stack. The Stack Realproc stkreal is made up of three modules CTR, MEM, and SCTL. Here, process MEM is defined mutually recursively with another process MEM1. We assume that in this design all the modules share a global clock.

The realization uses a memory and a counter to (respectively) hold the stack locations and the stack pointer. A controller decodes the external commands and appropriately sequences the submodules. This design implements an unbounded stack. Operation *push* is implemented by incrementing the counter and writing into the location of the memory pointed by the counter. *pop* is implemented by decrementing the counter. *top* is implemented by reading the memory at the location pointed by the counter. Finally, *sdef* is implemented by doing nothing. Suitable control wire encodings trigger these operations.

The Submodule Behaviors

We first examine the details of the *write* and *read* operation of the MEM submodule. Write is invoked by event Iwrite. At this time, the address and data are to be held stable on the ports ?cdo and ?din respectively. One tick later MEM returns to its control state with data

```
--------------------- An up/down Counter ---------------------

CTR [cs] <=   Icdef, !cdo=cs    -> CTR [cs]
            | Iload, vdin=?cdi -> CTR [vdin]
            | Iup, !cdo=cs      -> CTR [up(cs)]
            | Idown, !cdo=cs    -> CTR [down(cs)]

--------------------- A read/write Memory ---------------------

MEM  [ms]  <=   Imdef                       -> MEM [ms]
              | Iwrite, va=?cdo, vd=?din -> MEM [write(ms,va,vd)]
              | Iread, va=?cdo            -> MEM1[ms, va]

MEM1 [ms,oa] <=
           Imdef, !dout=read(ms,oa)          -> MEM [ms]
         | Iwrite, na=?cdo, vd=?din,
                  !dout=read(ms,oa)          -> MEM [write(ms,na,vd)]
         | Iread, na=?cdo,!dout=read(ms,oa)  -> MEM1[ms, na]

--------------------- A stack controller ---------------------

SCTL <=    Isdef, Omdef, Ocdef                                  -> SCTL
         | Ireset, Omdef, Ocdef -> Oload, Omdef                 -> SCTL
         | Ipush, Omdef, Ocdef  -> Oup, Omdef -> Owrite, Ocdef -> SCTL
         | Ipop, Omdef, Ocdef   -> Odown, Omdef                 -> SCTL
         | Itop, Omdef, Ocdef   -> Ocdef, Oread -> Oidle        -> SCTL

--------------------- The Stack Realization ---------------------

stkreal [cs,ms] <= Hide {load,up,down,cdef,read,write,mdef,cdo} in
                  CTR [cs] || MEM [ms] || SCTL
```

Figure 13: Specifications of the Submodules of the stack

```
PCTL <=  Isdef, Omdef, Ocdef                          -> PCTL
       | Ireset, Omdef, Ocdef -> Oload, Omdef         -> PCTL
       | Ipush, Omdef, Ocdef  -> Oup, Omdef           -> write, PCTL
       | Ipop, Omdef, Ocdef   -> Odown, Omdef         -> PCTL
       | Itop, Omdef, Ocdef   -> Ocdef, Oread -> Oidle -> PCTL
```

Figure 14: The Specification of a Pipelined Stack Controller

path state write(ms,va,vd).

The implementation of the *read* operation is trickier. We want to exploit a degree of pipelining afforded by the presence of the memory data register in this MEM. Specifically, consider two read operations issued one after the other. While we are collecting the results of the first read, we wish to start the second read. (Without this, read will cost us two cycles.) So starting from MEM in data path state ms, when Iread is invoked with address input ?cdo held stable, the MEM process turns itself into the MEM1 process. The MEM1 process awaits the operations *read*, *write* and *mdef while* outputting the result of the previous *read* on port !dout. While *reads* keep coming, we stay in state MEM1. When something other than *read* comes, we go back to state MEM.

Implementation of *push*

Now consider how *push* is implemented by focusing on SCTL and seeing what it does on the data path modules. When SCTL decodes the Ipush command, it outputs Omdef and Ocdef to the memory and the counter, thereby keeping both MEM and CTR inactive. In the next step it outputs Oup and Omdef thereby incrementing the counter, keeping the memory unchanged. In the next step it outputs Owrite and Ocdef thereby writing into the memory, at the address pointed to *by the incremented counter value* the data item that is *now asserted* at the input ?din. SCTL then goes back to its top-level control state. All the other operations are implemented similarly.

3.3 Pipelining the Stack

After obtaining a realization, the designer is usually interested in optimizing the design either manually or automatically. Pipelining, or *overlapping the internal activities within a system*, is a frequently adopted optimization. (There are other optimizations such as the sharing of ALUs, busses, etc.; we do not consider these in this paper.) Once manually pipelined, it is necessary to validate the functional correctness of the system.

It turns out that we can pipeline stkreal to a large extent. We illustrate this by "pipelining the *push* operation", *i.e.* carrying over some of the computation associated with *push* into the following operation. To see how this can be done, consider how *push* was implemented by SCTL. At first, SCTL performed the *up* operation on the counter. It then performed the *write* operation on the memory and *only then* did it return to its top-level control state. *However* SCTL could have been *waiting for the next operation while the* write *operation was still in progress internally.* (As we show in section 4, this wasted period does show up as an extra idling step in the behavior deduced by PARCOMP.) The pipelined controller PCTL given in figure 14 achieves this degree of pipelining. If it is used in lieu of SCTL, we get a realization called **pstkreal**. By introducing PCTL in lieu of SCTL we increase the number of control states in the controller in return for the increased speed of operation. Let us examine how **pstkreal** performs *push*.

pstkreal first decodes the command Ipush. Then it increments the counter via Oup. It then turns itself into the process write,PCTL. *This is a controller similar to PCTL with the difference that while waiting for the next operation it keeps MEM busy internally by applying the* Owrite *event on it.*

3.4 Questions to be Addressed

Having specified **stkabs**, **stkreal** and **pstkreal**, the following questions arise naturally:

- *Deducing Behavior from Structure:* Can we automatically infer a single process equivalent to **stkreal** as well as **pstkreal**? What are some applications of this PARallel COMPosition algorithm?
- *A Verification Problem:* Is **stkreal** a correct design corresponding to the requirements *expressed in* **stkabs**? *How do we determine this?*
- Given the controller in figure 14 and given the claim that this controller correctly pipelines the stack, how do we verify this claim? In what sense(s) is(are) a pipelined system comparable to a non-pipelined system?
- *Specification Directed Testing:* How do HOP specifications help in testing systems?
- What is a *system design methodology* using HOP?

These questions are answered in the following sections.

4 Verification Criteria, and Illustration Thereof

The denotation of a HOP process is its *trace-nodebinding* (TN) set. Therefore two HOP processes are equivalent if they have identical TN sets. However in many practical situations, two processes that are equivalent in many useful senses *do not have identical TN sets.* As we will soon show, a pipelined hardware design can contain different traces than present in either the requirements specification or a non-pipelined design.

In this section we address the verification of HOP processes. We illustrate it on the non-pipelined realization of the stack. Then we consider the problems that arise in the verification of pipelined hardware, and suggest possible solutions.

4.1 The Non-pipelined Stack Realization

We consider the stack realization **stkreal** of figure 13. The behavior of **stkreal** with respect to its external ports and external events can be inferred using PARCOMP. The details of this procedure will be provided in section 5. The inferred process, STKPAR, is shown in figure 15. Also shown in this figure for easy reference are: (a) PSTKPAR, the inferred behavior of the pipelined stack; (b) **stkabs**, the requirements specification of the stack.

```
stkabs [dps] <=
   Ireset ~> Ofree -> stkabs[reset(dps)]
 | Ipush  ~> Idata_avail, vdin = ?din ~> Ofree -> stkabs[push(dps,vdin)]
 | Ipop   ~> Ofree -> stkabs[pop(dps)]
 | Itop   ~> Otop_avail, !dout=top(dps) ~> Ofree -> stkabs[dps]
 | Isdef  ~> Ofree -> stkabs[dps]
%-----------------------------------------------------
STKPAR [cs,ms] <=
     Ireset -> Oidle, Ofree ->  STKPAR [0,ms]
   | Ipush -> Oidle -> Idata_avail, vdin=?din, Ofree
                        -> STKPAR [up(cs), write(ms,up(cs),vdin)]
   | Ipop  -> Oidle, Ofree ->  STKPAR [down(cs), ms]
   | Itop  -> Oidle -> Otop_avail, Ofree, !dout=read(ms,cs) -> STKPAR [cs,ms]
   | Isdef, Ofree ->  STKPAR [cs,ms]
%-----------------------------------------------------
PSTKPAR [cs,ms] <=
     Ireset -> Oidle, Ofree ->  PSTKPAR [0,ms]
   | Ipush -> Oidle, Ofree -> PSTKPAR1 [ up(cs), ms ]
   | Ipop  -> Oidle, Ofree ->  PSTKPAR [down(cs), ms]
   | Itop  -> Oidle -> Otop_avail, Ofree, !dout=read(ms,cs)
           -> PSTKPAR [cs,ms]
   | Isdef, Ofree ->  PSTKPAR [cs,ms]

PSTKPAR1 [cs1, ms1] <=
     Ireset, Idata_avail, vdin=?din -> Oidle, Ofree
           -> PSTKPAR [0, write(ms1,cs1,vdin)]
   | Ipush, Idata_avail, vdin=?din -> Oidle, Ofree
                        -> PSTKPAR1 [up(cs1), write(ms1,cs1,vdin)]
   | Ipop, Idata_avail, vdin=?din  -> Oidle, Ofree
           -> PSTKPAR [down(cs1), write(ms1,cs1,vdin)]
   | Itop, Idata_avail, vdin=?din
           -> Oidle, Ofree
           -> !dout=read(write(ms1,cs1),cs1), Otop_avail, Ofree
           ->  PSTKPAR  [cs1, write(ms1,cs1,vdin)]
   | Isdef, Idata_avail, vdin=?din -> Oidle, Ofree
           -> PSTKPAR [cs1, write(ms1,cs1,vdin)]
```

Figure 15: stkabs, STKPAR and PSTKPAR

Let us examine the push operation of STKPAR. It is initiated by Ipush. After this, during the second time step, the user sees STKPAR idling. Actually at this time the counter module CTR is getting incremented by one, but the 'up' event on CTR is hidden and hence not visible outside. During the third time step, *write* is performed on MEM. As a consequence, port ?din is being sampled now. Starting from the fourth tick, STKPAR continues to behave as before, with its data path state advancing to the pair of states $< up(cs), write(ms, up(cs), vd) >$. The events Idata_avail and Ofree are added in by the user as will be explained shortly.

Now consider operation *top*. During the second time step, a 'read' is issued on MEM, but it appears as Oidle because 'read' is hidden. The result of *top* becomes available during the third time step.

Adding-in Conceptual Events to an Inferred Absproc

A design is a more detailed implementation that has to relate to its requirements specification. Different designs however have different *usage protocols*. The protocols are to be made explicit before we can meaningfully compare a design with a requirements specification. In our approach this is done by adding-in events that characterize the protocol obeyed by the absproc description *at the appropriate instants* of the inferred absproc. In the STKPAR specification in figure 15, the events Idata_avail, Ofree, and Otop_avail are added in by the user for this purpose. For example, in the *push* operation of STKPAR, Idata_avail is generated during the third time instant.

4.2 Our Verification Technique

Currently our formal verification methodology applies to those systems whose operations (modes of behavior) can be viewed as a "constructor-observer" (*c_o*) experiment. A process $P[S]$ subject to a *c_o* experiment consumes a sequence of input values I_i, produces a sequence of output values O_j and turns into a process $P[Cons(S, I_i)]$. It is assumed that the output O_j can be modeled using an *observer function application* of the form $Obs(S, I_i)$. Likewise, it is assumed that data path state changes can be captured by a *constructor function application* of the form $Cons(S, I_i)$. *constructor experiments* and *observer experiments* are special cases of c_o experiments where either a new data path state or a new output port value gets created (but not both). *c_o* experiments start with distinct *command events*, (such as Ipush).

A large number of digital systems can be viewed in this manner. For instance, in the proof of correctness of a simple microprocessor reported in [Coh88], only *constructor* and *observer* experiments are considered. This is also the case in the proof of correctness of yet another microprocessor in [Hun87]. In both these works the constructor experiments correspond to the system state changes caused by the execution of instructions, and an observer experiment consists of the observation of the register values attained in-between constructor experiments. In both these works, the proof involves showing that if the systems specified at the requirements and the design levels start in two observationally equivalent states S and S' and are subject to the same macro-instruction, the systems wind up in two states $S1$ and $S1'$ that are also observationally equivalent. Our technique is related to these works. Due to the usage of a process model in HOP, we do not have to introduce "oracles" [Hun87] to capture external input that may arrive at unspecified time instants. HOP's "busy wait" (~> Idata_avail,vdin:=?din) captures this effect. If necessary, we can introduce "tester processes" to model the world external to a HOP process in as much detail as we wish.

4.3 An Outline of the Verification of STKPAR

Let us first consider a system with only constructor and observer experiments—in this case, the stack. We pick a constructor experiment ce_i and subject stkabs and STKPAR to it. We then follow ce_i with an observer experiment oe_j. We then assert that the values returned by the latter must agree, thus obtaining a Verification Condition (VC). In this way, we consider all possible sequences of constructor experiments and observer experiments.

Though this may sound like performing an infinite number of proofs, we can achieve the same effect by performing a finite number of proofs, using a form of *data type induction* as applied to HOP processes. In HOP, data path states are modeled using constructor functions that are defined *equationally* using data type axioms. Similarly, port values are modeled using observers which are also defined via data type axioms. We will take advantage of these facts in our verification methodology.

We assume that processes stkabs and STKPAR, when started in data path states *dps* and $< cs, ms >$ are *observationally equivalent*. By this, we mean that there exists no observer experiment that can distinguish between these two processes. (In our example, performing *top* results in the same value being produced on the port !dout.) We then show that performing the same constructor experiment ce_i on $stkabs[dps]$ and $STKPAR[cs,ms]$ results in processes $stkabs[dps']$ and $STKPAR[cs', ms']$ *that are also observationally equivalent*.

Consider the *top* and *push* operations. Applying our methodology results in the following VC:

$$\frac{top(dps) = read(ms, cs)}{top(push(dps, vdin)) = read(write(ms, up(cs), vdin), up(cs))}.$$

The antecedent part is obtained from the assumption that the processes start by being observationally equivalent-meaning that *top* must not distinguish them. The consequent is obtained by applying a *top* first followed by a *push*.

This VC can be shown to be valid using the equational properties of the *stack*, *memory* and *counter abstract data types*. A typical proof would consist of unfolding the consequent and performing a case analysis, where the cases considered follow from the antecedent. In the current example, familiar stack and memory axioms allow us to immediately reduce the consequent to the tautology $vdin = vdin$. We then repeat the procedure for all the other constructors *pop*, *reset*, and *sdef*.

We handle systems with c_o experiments as follows. For a c_o experiment *op*, we first identify its associated constructor and observer functions C_{op} and O_{op}. When *op* is used as the final experiment in a sequence of experiments, we would get VCs of the following form:

$$\frac{O_{op.abs}(C_{op.abs}(dps)) = O_{op.real}(C_{op.real}(DPS))}{O_{op.abs}(C_{op.abs}(D_{abs}(dps))) = O_{op.real}(C_{op.real}(D_{real}(DPS)))}.$$

In this VC, the subscript *.abs* denotes "as defined at the requirements level". The subscript *.real* is meant to denote "the expression obtained from the realproc level". D is a constructor differen' from C, and results from either a *constructor* experiment or a c_o experiment. *dps* and *DPS* are the data path states at the requirements and design specification levels respectively.

Our assumption that the modes of behavior can be thought of as *c_o* experiments imparts a structure to TN sets such that every trace and node-binding sequence is a concatenation of the traces and node-bindings of the individual *c_o* experiments. The equivalence induced on TN sets by our verification approach is captured by the following properties:

- STKPAR and stkabs have the same trace sets if all intervening Oidle events are dropped.
- For every data assertion in stkabs, STKPAR also makes those assertions. (The opposite need not be true.)
- For every data query made in stkabs, STKPAR also makes the same query, with the same data input provided in both cases.

4.4 Verification of PSTKPAR

Similar to STKPAR, we obtain PSTKPAR through PARCOMP. The specification is in figure 15. It includes the events Idata_avail, Ofree, and Otop_avail added in by the designer, for reasons to be explained.

Consider the implementation of *push*. In stkabs, it consists of the sequence

```
Ipush -> Idata_avail, vdin:=?din -> Ofree -> stkabs[push(dps,vdin)]
```

whereas in PSTKPAR, it consists of the sequence

```
Ipush -> Ofree -> PSTKPAR1[up(cs),ms]
```

followed by the sequence generated while executing PSTKPAR1. Regardless of the choice offered to PSTKPAR1, it awaits Idata_avail at the first step.

Observe that the ordering of events Ipush -> Idata_avail -> Ofree changes to Ipush -> Ofree -> Idata_avail in PSTKPAR. PSTKPAR issues Ofree one step earlier than STKPAR to permit the next operation to begin. Thus the traces of PSTKPAR and stkabs are not the same.

To complicate things further, PSTKPAR supports the same set of experiments, but in a different way. Consider the sequence of operations Ipush,Itop and consider how PSTKPAR of figure 15 would execute it. Two ticks after Ipush, we end up in control state PSTKPAR1 where the choice Itop is accepted. Strictly speaking, a "*top* experiment" begins at control state PSTKPAR1—*however* while this experiment goes on, the state of MEM is getting updated, with the result that *top* doesn't appear to be an *observer experiment*.

Our solution to these problems is based on the following assumptions:

- Only the sequential ordering of the *command events* present in the trace sets need agree. Other events that are related to data availability and modules becoming free can be ignored while comparing trace sets.
- Assume that every pipelined system has a "nop" event that can be inserted in between two pipelined operations to get rid of the effect of pipelining. It is possible to artificially introduce such an operation if it does not exist. For the stack, *sdef* is such an operation that already exists.

Consider the only pipelined operation *push* of the stack. (All such pipelined operations are considered in general.) To establish whether *push* has been correctly implemented, we should try to establish the following equivalences between sequences of operations: For all OP other than *push*:

- $push;\ sdef;\ OP \equiv push;\ OP;\ sdef$
- $push;\ sdef;\ push;\ OP \equiv push;\ push;\ OP;\ sdef$

The first line says that doing the sequence of operations shown on the left or on the right *on the very-same pipelined system* should be treated as being equivalent by all ensuing operations. The rationale behind choosing the first sequence is the following. The left-hand side of the first sequence pads a *sdef* in between *push* and OP, thus studying the implementation of *push* without pipelining. The right-hand side checks the effect of *push* when part of the computation of *push* is allowed to overlap into OP.

Section Summary

We believe that the use of abstract data types as well as the absence of nondeterminism contributes to the simplicity of the verification of HOP specifications. Our approach to verification has similarities to that reported in [Coh88], [Hun87] and [GSS87]. None of these works consider pipelined modes of behavior.

Pipelined modes of behavior are very commonly employed in high-performance microprocessors. Often pipelining is not fully automated, and so results in the introduction of many sequencing errors in the initial designs [Wei87]. Since formal verification is not employed in practice, there is a great danger that some sequencing errors remain even in fabricated chips despite extensive simulations. To our knowledge no one has considered the verification of pipelined hardware systems, except in limited domains such as systolic systems [Hen84]. This, in our opinion, is an important area of new research.

5 The Basic PARCOMP, and PARCOMP-DC

The operational rules of HOP permit us to simplify the definition of a collection of process P_i involving the $\|$, **Renaming** and **Hiding** operators into the definition of a single process P' where (i) P' does not contain any occurrences of the **Renaming** or **Hiding** operators; (ii) P' has the $\|$ operators pushed "deep inside it"; (iii) P' does not have any data queries or assertions in its body; instead, for every data query/assertion pair $< dq, da >$ present in P, P' has a *functional expression* in its body. Further, the collection P_i and P' have identical TN sets with respects to their external ports—a fact we have already exploited during formal verification.

This procedure called PARCOMP is well defined *i.e.* effective. It always terminates because:

- The operational rules always effect simplification under a well-founded ordering; this is true because each operational rule considered considers a process P of the form $ca \rightarrow P'$, and the rule is then recursively applied to P';

- When we encounter a form $P[sp] \parallel ... \parallel Q[sq]$ (where sp and sq are *terms*) for the first time, instead of unraveling this form through the rules of HOP, we unravel a more general form $P[x] \parallel ... \parallel Q[y]$ where x, y are variables;
- A collection P_i introduces a finite number of processes through mutual recursion. As the $\parallel$ operator is "moved inwards", eventually there will come a stage where we will *re-encounter* expressions of the form $P[sp] \parallel ... \parallel Q[sq]$. Regardless of what sp and sq are, when we re-encounter a form, we need not re-explore the form $P[sp] \parallel ... \parallel Q[sq]$ *because* a most general expansion for this form has already been obtained for the case where sp and sq are variables.

Hence PARCOMP is an algorithm.

In order to determine how efficient and useful PARCOMP is in practice, we coded it and tried it out on a number of examples. We conclude that: (i) it is efficient enough to be used for many purposes; (ii) the single process inferred by PARCOMP is attractive in many ways:

- *It is easier to understand* because the internal details are hidden;
- *It is and more efficient to simulate* because internal value communications appear as functional expressions in the inferred specification; therefore we need not maintain information regarding port value bindings during simulation;
- *Synchronization errors* can be detected because synchronization errors usually result in PARCOMP generating a finite process;
- *Only the useful modes of behavior are retained.* In particular, the behavioral description of all the "idle hardware" is not retained. Idle hardware includes both *unused* and *under-utilized* hardware—*i.e.* modules with only part of their operations used.

5.1 Steps in the Basic PARCOMP Algorithm

We consider a collection of processes that do not contain any *Cond-processes* (a more general definition of PARCOMP is given in Appendix C). All required renamings are assumed to be already done. We are given the process descriptions of the processes to be composed, as well as the *hiding set* HS containing events and ports that are hidden. Then, the steps in PARCOMP for this problem are:

1. Start all the N processes in their respective starting states.

2. March the processes in unison (lockstep-synchronously) until all N-tuples of control states *that are equidistant from the starting state* have been visited. (A node is at distance d from the start state if it can be reached via d transitions from the start state.) Record each such N-tuple of control states visited as a control state of the inferred process.

3. In moving from the control state N-tuple S_x to the control state N-tuple S_y, the following actions are taken:

   (a) Collect the actions labeling the transitions going from the ith component of S_x to the ith component of S_y, for all i in N. Call this collection C_{xy}.

(b) Reduce C_{xy} by applying the action product operator "," to its members. Obtain the *normal form* D_{xy} such that any pair of elements in D_{xy} is irreducible under ",".

(c) If there are unsynchronized actions in D_{xy} that are in the hiding set HS, then mark state S_y as un-reachable from S_x.

(d) Replace the synchronized events of D_{xy} that are in HS by $\overline{idle}$.

(e) Collect the result of value communications as value bindings to the variables in the data query:

   i. Represent these bindings as *let* blocks corresponding to the state S_y.

   ii. For multiple writers on a node, apply the *bus* function to the data assertions to determine the resultant value on the node.

   iii. Since the user is required to indicate each tristatable port in his system, bus connections of non-tristate ports can be caught.

(f) Recursively call PARCOMP starting at S_y.

4. In the end, remove states that are unreachable from anywhere.

5. If the resulting process is a finite proess, then discover where it becomes STOP, and for all those points flag a sequencing error, and generate diagnostic information.

5.1.1 Illustration of PARCOMP on the *push* Operation of stkreal

Consider the controller `SCTL`, counter `CTR` and memory `MEM` of figure 13 to be in their starting control states. Let us march them around from control state $< SCTL, CTR[cs], MEM[ms] >$ back to $< SCTL, CTR[cs'], MEM[ms'] >$.

1. The first group of actions to be considered are
   `Ipush, Omdef, Ocdef, Imdef, Icdef`. This simplifies to
   `Ipush, Smdef, Scdef`, and after applying hiding, becomes
   `Ipush` .

2. The second group of actions are
   `Oup, Omdef, Iup, !cdo = cs, Imdef`
   which after simplification and applying hiding becomes
   `Oidle`.

3. The third group of actions are
   `Owrite, Ocdef, Iwrite, va=?cdo, vd=?din, Icdef, !cdo=cs`
   and after simplification and hiding becomes
   `vd=?din`.

4. We then move back to $< SCTL, CTR[up(cs)], MEM[write(ms, up(cs), vd)] >$.

5. There are no other paths that survive for the *push* operation. For instance, the transition labeled by

`Ipush, Ocdef, Iup, Omdef, Iwrite, ..`

The pairs S_0, T_1 and S_1, T_0 need not be considered.

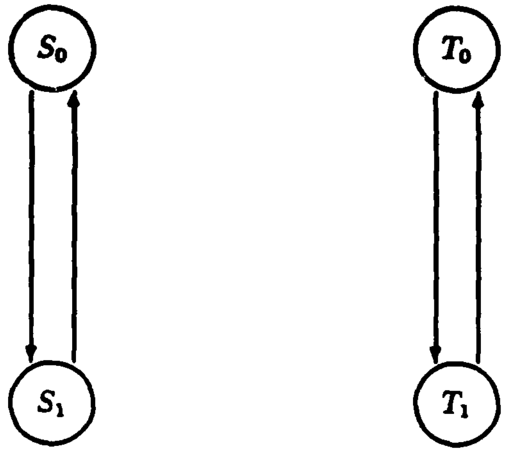

Figure 16: Illustrating Marching in Unison

is not a feasible transition because while the stack controller is trying to apply the `Ocdef` and `Omdef` events to the counter and memory respectively, the counter and memory themselves are awaiting `Iup` and `Iwrite`. Unsynchronized as well as hidden transitions get pruned.

5.1.2 Heuristics Employed by PARCOMP

- Generating the N-tuples of states by taking a cross-product is wasteful, as figure 16 shows. We do it by *marching in unison*; its results are also shown in figure 16.
- A hidden unsynchronized event labeling a transition essentially removes the transition from the graph. Using this mechanism, many control state N-tuples become un-reachable and are removed early.

5.1.3 Applications of PARCOMP

- Obtaining simpler behaviors during design entry through a schematic entry system. This way after entering a schematic, the behavior can be inferred, validated, and made a part of the module library for further upwards compositio.
- Simulation can be easily achieved by introducing a *tester process* similar to that used by [Mil85b]. The tester process is composed with the system to be tested and the

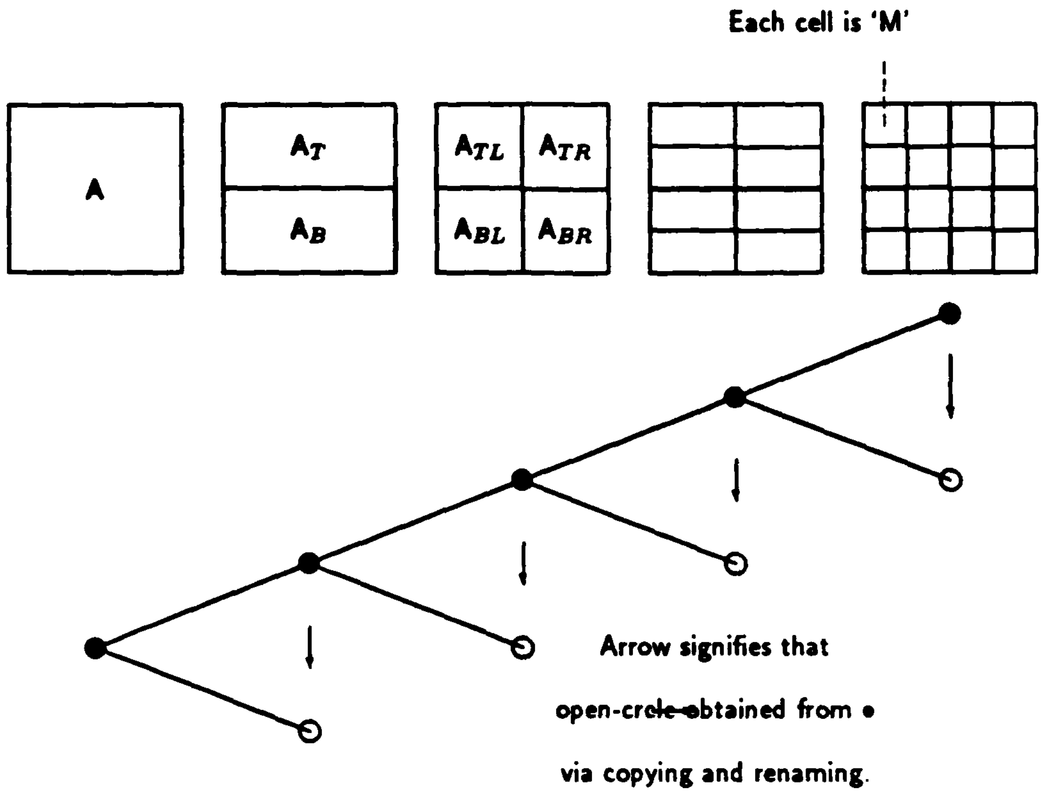

Figure 17: Divide and Conquer PARCOMP

resultant process can then be run.

- Symbolic execution of the inferred behavior is possible.
- The inferred behavior can be used for formal verification.

5.2 A Divide and Conquer Version of PARCOMP

PARCOMP-DC is a potentially faster way of computing PARCOMP by exploiting the geometrical regularity of arhythmic arrays. We will take a generic arhythmic array structure and illustrate PARCOMP-DC on it. Due to the shortage of space, we relegate a more interesting example—the LRU matrix—to Appendix A. In this section we derive an expression for the computational savings possible due to PARCOMP-DC.

Consider the array A shown in figure 17. It consists of a collection of modules M connected in a regular interconnection pattern. For simplicity assume a nearest-neighbor connection that is regular in both the dimensions.

Consider the problem of computing $PARCOMP(A)$; *i.e.* the composition of all the Ms constituting A. $PARCOMP$ is both commutative and associative. Hence, we can split A into two halves, say A_T standing for "the top of A" and A_B, standing for "the bottom of A". Thus,

$$PARCOMP(A) = PARCOMP(\ PARCOMP(A_T),\ PARCOMP(A_B)\ ).$$

But $PARCOMP(A_B)$ is easily obtained from $PARCOMP(A_T)$ by renaming the ports of A_T to the corresponding ports of A_B. Thus we need compute *only* $PARCOMP(A_T)$ using the $PARCOMP$ procedure; we can then obtain $PARCOMP(A_B)$ by making a copy of the data structure that represents $PARCOMP(A_T)$, and apply suitable renamings to it.

But the process need not stop at the top-level of division. We can split A_T into A_{TL} (the "left half of A_T) and A_{TR} (the "right half of A_T) and again exploit the fact that A_{TR} can be obtained from A_{TL} through copying and renaming. This gives us a divide-and-conquer procedure. We depict the execution of this procedure as a tree in figure 17.

PARCOMP-DC is often more efficient than PARCOMP. Let us make an approximate cost analysis. The worst-case time complexity of PARCOMP is primarily dependent on the number of control states that we have in a process diagram. Specifically, it can be equal to the cross-product of the number of control states in each of the processes. Suppose for simplicity that array A is square, and has N modules of type M, M has C control states in it, and that N be a power of 2. Then

$$cost\_parcomp(A) = C^N$$

because we may, in the worst-case, end-up taking a full cross-product of the process diagrams of the N modules.

Suppose that the modules formed during the division process of PARCOMP-DC, M, ..., A_{TL}, A_T, A all have D control states "on the average"; more precisely D must be the *root mean square* value of the number of control states. Then

$$cost\_parcomp\_dc(A) = \log_2(N) \times D^2.$$

This is because we are doing $\log_2(N)$ PARCOMPs of two modules at a time, where each of these modules have, D control states as a *root mean square*. Root mean square is needed becaus we are squaring D within the summation (we are doing the summation $log_2(N)$ times). We assume (as is the case in our data structures) that copying and renaming a process description has negligible cost.

Firstly we note that D does not tend to increase as the size of the modules grow. In fact for the LRU module D was equal to C. Thus if D is close to C and if M is large, then there is a significant payoff by using PARCOMP-DC.

The behavior inferred by PARCOMP-DC for large arhythmic arrays is not very intuitively understandable for human readers. We show the result of doing one-level of PARCOMP for the LRU module in Appendix A. In conclusion, the following approach is suggested for handling arhythmic arrays:

- Perform PARCOMP of two modules of the array;
- Study the inferred behavior and see if it is verifiable manually or through exhaustive simulation; (for the LRU module, we discovered a sequencing error by the former technique.)
- Apply PARCOMP or PARCOMP-DC whichever is faster[4]. The behavior inferred by PARCOMP (or PARCOMP-DC) will have complex if-then-else functions. Construct tabular functions corresponding to these.
- Use these tabular functions for efficient simulation.
- Try to perform formal verification of the whole array by setting up an induction.

[4] We may race them and pick the winner!

6 Concluding Remarks

6.1 A Design Methodology Based on HOP

The following steps capture a top-down design methodology that is currently under investigation. While our investigation is still in its infancy, we believe it to be important to show how we have *fused* verification with design.

- Write the *requirements specification* for the module to be designed;
- Identify the submodules that are to constitute the realization;
- Write requirements specifications for the submodules;
- Apply PARCOMP to the requirements specifications of the submodules;
- Follow the HOP verification methodology and verify that the behavior inferred is equivalent to the original requirements specification;
- (Recursively) invoke the HOP design methodology on the submodules, thereby obtaining a *circuit* and a *design specification* for them;
- *Match* the requirements specification for the submodules against their corresponding design specifications; obtain a definition of the events in the requirements specification in terms of the events at the design level;
- Propagate this information back to the level of M thereby obtaining a design specification for M;
- Apply optimizing transformations to M.

In many of the above steps, we believe that a graphical editor for process diagrams can be gainfully employed. For instance, in many situations a user could graphically specify a highly inefficient but functionally correct controller, such as SCTL. After completing a first design based on it, the optimized version PCTL can be obtained. Some heuristics are known to us: *e.g.* "burping" all the $\overline{idle}$ steps in the inferred process by overlapping actions. In principle, the approach for pipelining calls for systematically rearranging events labeling process transitions without violating observational equivalence between requirements and design specifications. In this way we hope that the rigorous semantics of HOP would help in design synthesis and optimization.

6.2 Ongoing and Future Work

We have implemented a first prototype of PARCOMP to study its performance on simple examples. Based on this experience, we are now engaged in developing a more elaborate implementation of the HOP system. The implementation uses FROBS [Mue87] that supports object-oriented programming, data activated *daemons*, and an *inference engine*. This would help in watching simulation results in a very flexible manner. Complex trace mechanisms, such as in logic-state analyzers, can be built. We summarize some of our results to date in appendix B.

Concurrently we are engaged in the concurrent specification and design of a large ASIC called the RBC [FTG88b,FTG88a]. Tackling this large example has benefited HOP greatly. For example, several arhythmic arrays are present in the RBC. Many important issues such as modeling *through connections* satisfactorily, supporting grouping of ports into *arrays* and *records* of ports for convenience, etc. are quite important if we were to manage the complexity of a large specification. We are aiming towards a prototype of both the RBC as well as the HOP system.

A preliminary design of the RBC was verified by hand using the technique illustrated on the stack. Realizing the tedium and error-prone nature of the hand-proof, we plan to semi-automate some of the verification steps in the long run. We plan to study the equational laws of HOP as well as investigate notions of equivalence among HOP processes.

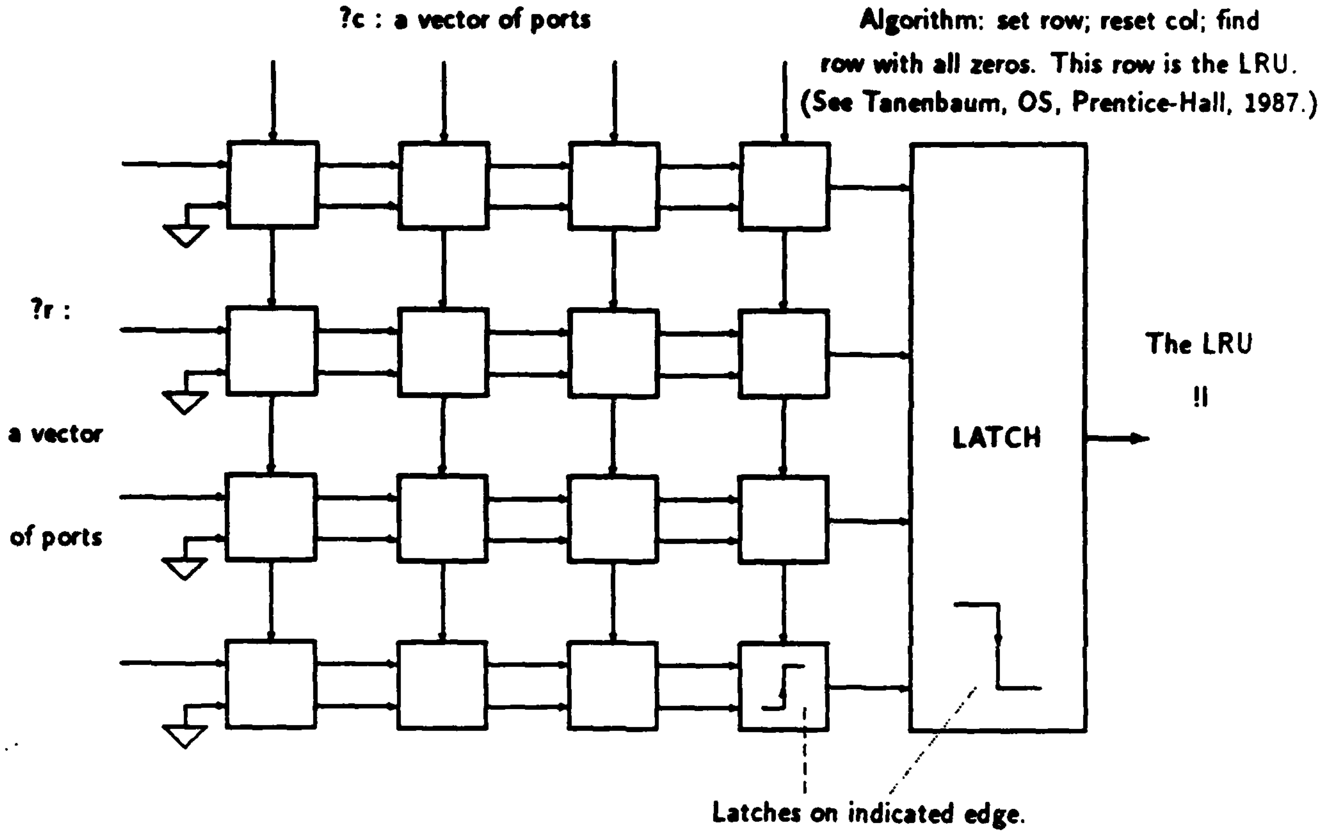
?c : a vector of ports
Algorithm: set row; reset col; find row with all zeros. This row is the LRU. (See Tanenbaum, OS, Prentice-Hall, 1987.)
?r :
a vector
of ports
LATCH
The LRU
!!
Latches on indicated edge.

Figure 18: An LRU Matrix

A An LRU Matrix

Figure 18 shows a unit to compute the "Least Recently Used" location, as described in [Tan87, page 217]. The specification of one cell of this unit, `lru_cell`, is shown in figure 19.

This unit is meant to operate inside an memory management unit as follows.

Conceptually the algorithm to be followed is the following two-phase algorithm. Initially the matrix starts with all zeros. Whenever a memory address is accessed, a unary representation of the address is fed to both the ?r and ?c inputs. The bits in the indexed row are first set. Thereafter the bits in the indexed column are reset. The LRU is always pointed by that row that has all zeros. After four distinct address accesses, there is guaranteed to be such a row. The implementation does not use the two-phase algorithm directly; rather priority logic resident in each `lru_cell` decide whether to set or to reset the cell. The description in figure 19 details the algorithm.

We performed one step of PARCOMP-DC by hand and obtained the description shown in figure 20 for the behavior of two cells acting together. This specification can be used for simulation and design verification. Following this, we can mechanically derive the behavior of the entire array, represent it as a tabular function and employ it for simulation.

```
-- This spec. clearly shows that lrucell changes its datapath
-- state during the rising edge of the clock and puts out a
-- value on port !nextout during the falling edge. Thus during the
-- falling edge, the external latch can latch in the results.
-- The ''if'' functions used herein have obvious implementations
-- using combinational logic. In addition, each lrucell has a one-bit
-- Flip-flop. Details available from the author upon request.
--
ABSPROC lrucell
 PORTS  ?previn, ?rowin, ?colin, !nextout : BIT
 CLOCKS singlephase(ck,not(ck))
 EVENTS ckrise = ck
        ckfall = not(ck)
 PROTOCOL
   lrucell [dps] <= ckrise, vcolin = ?colin, vrowin = ?rowin
                        -> lrucell1 [ if(vcolin,0,
                                        if(vrowin,1,dps)) ]
   lrucell1 [dps] <= ckfall, vprevin = ?previn,
                        !nextout = if(vprevin,1,dps) -> lrucell [dps]
END lrucell
```

Figure 19: Specification of the lru_cell module

```
--  Two lrucells as shown are subject to a PARCOMP. The result is shown as an absproc.
--  These ?colin ports are together regarded as a single VECTOR PORT ?colin.
--
--         ?colin[0]   ?colin[1]
--              |           |
--              v           v
--   ?rowin |-------|   |-------|
--      --->|       |-->|       |   !nextout
--          | dps[0]|   | dps[1]|--->
--        ->|       |-->|       |
--        | |-------|   |-------|
--        |
--        gnd (internally grounded)
--
ABSPROC twolcs
 PORTS  ?rowin, ?colin[0], ?colin[1], !nextout : BIT
 CLOCKS singlephase(ck,not(ck))
 EVENTS ckrise = ck
        ckfall = not(ck)
 PROTOCOL
   twolcs [ dps[0], dps[1] ] -- Fully expanded form dps[0],[1] is for convenience
   <= ckrise, vcolin = ?colin, vrowin = ?rowin
       -> twolcs1  [ if(vcolin[0],0,
                        if(vrowin,1,
                             dps[0])),
                     if(vcolin[1],0,
                        if(vrowin,1,
                             dps[1])) ]
   twolcs1 [ dps[0], dps[1] ]
   <= ckfall, !nextout = if(vcolin[0], if(vcolin[1],0,
                                          if(vrowin,1,dps[1])
                                         ),
                            if(vrowin,1,
                                if(dps[0],1,
                                   if(vcolin[1],0,
                                      if(vrowin,1,dps[1])
                                     )))
                           ),
        twolcs[dps[0], dps[1]]
--
-- These if forms may be converted into a tabular function that is
-- sequentially searched during simulation. So the simulator doesn't
-- have to keep track of the internal wires of the LRU matrix, and
-- hence becomes efficient.
END lrucell
```

Figure 20: Specification of the lru_cell module

B Results To Date

- An initial design of the RBC chip has been verified by hand.
- PARCOMP has been coded in Lisp. Applying it to the stack example considered in this paper resulted in the inferred absprocs that we have shown. The execution time was in the range of a few seconds.
- Other modules specified in HOP include a Translation Lookaside Buffer and the internal circuitry of the RBC chip that consists of about ten large arhythmic arrays.

C A Specification of PARCOMP

$\boxed{\textit{Input:}}$ An expression $\mathtt{Hide}$ HS *in* $\| \{P_i[\overline{X_i}], ..., C_j[\overline{X_j}], ...\}$ for $i \in \{1..m\}, j \in \{1..n\}$. C_j are conditional processes of the form $C_j[\overline{X_j}] = \mathtt{if}\ q_j\ \mathtt{then}\ T_j[g_j(\overline{X_j})]\mathtt{else}\ F_j[h_j(\overline{X_j})]$ and P_i are non-conditional processes of the form $P_i[\overline{X_i}] = y_i\ :\ initials_i \rightarrow R_i(y_i)$; Each P_i offers a set of initial choices $initials_i$ and for each choice y_i that is offered, the future behavior of P_i is $R_i(y_i)$. HS is the *Hidden Set*, the set of events and ports hidden from the parallel composition.

$\boxed{\textit{Output:}}$ A behaviorally identical process $P[\overline{X_i}, ..., \overline{X_j}, ...]$.

$\boxed{\textit{Method:}}$ A *done-list* is maintained for each parallel composition $\| \{P_i[\overline{X_i}], ...\}$ that has already been computed. Upon getting a call for performing parallel composition, the *done-list* is first consulted.

- If the requested parallel composition is in the *done-list*, return. Else enter it in the *done-list* and proceed as follows.
- Combine all conditional processes into one conditional process C. Combining two conditional processes is done as follows:

$$C_1[\overline{X_1}] = \mathtt{if}\ q_1\ \mathtt{then}\ T_1[g_1(\overline{X_1})]\ \mathtt{else}\ F_1[h_1(\overline{X_1})]$$

$$C_2[\overline{X_2}] = \mathtt{if}\ q_2\ \mathtt{then}\ T_2[g_2(\overline{X_2})]\ \mathtt{else}\ F_2[h_2(\overline{X_2})]$$

$$\begin{aligned} C_1[\overline{X_1}] \parallel C_2[\overline{X_2}] \quad = \quad & \mathtt{if}\ (q_1 \wedge q_2)\ \mathtt{then}\ T_1[g_1(\overline{X_1})] \parallel T_2[g_2(\overline{X_2})] \\ & \mathtt{else\ if}\ (q_1 \wedge not(q_2))\ \mathtt{then}\ T_1[g_1(\overline{X_1})] \parallel F_2[h_2(\overline{X_2})] \\ & else\ \ ...etc.\ (all\ four\ combinations) \end{aligned}$$

- Now we are left with the task of computing $\mathtt{Hide}$ HS *in* $\| \{P_i[\overline{X_i}], ..., C\}$. Let C be of the form

$$\mathtt{if}\ q_1\ \mathtt{then}\ C_1[g_1(\overline{X_1})]\mathtt{else\ if}\ q_2\ \mathtt{then}\ C_2[g_2(\overline{X_2})]etc.$$

  $\| \{P_i[\overline{X_i}], ..., C\}$ reduces to a conditional process with q_i as the conditions. This conditional has in it parallel compositions of the form $\| \{P_i[\overline{X_i}], ..., C_i\}$. that is (recursively) computed. Eventually we are faced with composing non-conditional processes in parallel. We take this up next.
- Consider $\| \{P_i[\overline{X_i}], ...\}$. Let each P_i be

$$\begin{aligned} P_i[\overline{X_i}] \quad = \quad & ca_i^1 \rightarrow R_i^1[f_i^1(\overline{X_i})] \\ | \quad & ca_i^2 \rightarrow R_i^2[f_i^2(\overline{X_i})] \\ | \quad & ... \\ | \quad & ca_i^{n_i} \rightarrow R_i^{n_i}[f_i^{n_i}(\overline{X_i})] \end{aligned}$$

-
- Generate tuples

$$T = < ca_1^{x_1}, ca_2^{x_2}, ...ca_m^{x_m} >$$

i.e. a tuple of the x_1th initial compound action offered by P_1, the x_2th initial compound action offered by P_2, etc. This tuple T is assumed to be the irreducible form arrived at after applying the action product rules of figure 8. According to the rule for parallel composition *Parcomp* all such tuples would become the initial choices of the resultant process. Following such choices, the resultant process would continue to behave like $\| \{R_1^{x_1}[f_1^{x_1}(\overline{X_1})]R_2^{x_2}[f_2^{x_2}(\overline{X_2})], \ldots\}$. However using the hiding information HS, we can prune many of these choices. In particular,

  - those tuples T that contain *unsynchronized* events e or $\overline{e}$ that belong to HS are dropped, and the corresponding arm of the synchronization tree is pruned;
  - those tuples T that contain *synchronized* events $\overline{\overline{e}}$ that belong to HS are replaced by $T[\overline{idle}/\overline{\overline{e}}]$.

- In computing

$$\| \{R_1^{x_1}[f_1^{x_1}(\overline{X_1})], R_2^{x_2}[f_2^{x_2}(\overline{X_2})], \ldots\},$$

the bindings generated by taking action products of the members of T are taken into account. Specifically, we construct a *let* block containing these bindings. □

References

[BP88] Graham Birtwistle and P.A.Subrahmanyam. *VLSI Specification, Verification and Synthesis*. Kluwer Academic Publishers, Boston, 1988. ISBN-0-89838-246-7.

[Bry84] Randall E. Bryant. A Switch Level Model and Simulator for MOS Digital Systems. *IEEE Transactions on Computer*, C-33:160–177, February 1984.

[BW] Alan B.Davis and Uri Weiser. On Modeling Regular Arrays. (This is not an exact reference.).

[CGM86] Albert Camilleri, Michael C. Gordon, and Tom Melham. Hardware Specification and Verification using Higher Order Logic. In *Processings of the IFIP WG 10.2 Working Conference on "From HDL Descriptions to Guaranteed Correct Circuit Designs", Grenoble, August 1986*, North-Holland, 1986.

[Coh88] Avra Cohn. A Proof of Correctness of the VIPER Microprocessors: The First Level. In Graham Birtwistle and P.A.Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 27–71, Kluwer Academic Publishers, Boston, 1988. ISBN-0-89838-246-7.

[FTG88a] Richard Fujimoto, Jya-Jang Tsai, and Ganesh Gopalakrishnan. Design and Performance of Special Purpose Hardware for Time Warp. In *The Computer Architecture Conference, Honolulu*, 1988. (Accepted for publication).

[FTG88b] Richard Fujimoto, Jya-Jang Tsai, and Ganesh Gopalakrishnan. The Roll Back Chip: Hardware Support for Distributed Simulation Using Time Warp. In *The Society for Computer Simulation Multiconference, San Diego, CA*, February 1988. (To Appear).

[GHM78] John V. Guttag, Ellis Horowitz, and David R. Musser. Abstract Data Types and Software Validation. *Communications of the ACM*, 21(12):1048–1064, December 1978.

[Gop] Ganesh C. Gopalakrishnan. The Specification and Verification of the Roll Back Chip. Available Upon Request From the Author.

[Gop86] Ganesh C. Gopalakrishnan. *From Algebraic Specifications to Correct VLSI Systems*. PhD thesis, Dept. of Computer Science, State University of New York, December 1986. (Also Tech. Report UU-CS-86-117 of Univ. of Utah).

[Gop87] Ganesh C. Gopalakrishnan. Synthesizing Synchronous Digital VLSI Controllers Using Petri nets. In *International Workshop on Petri Nets and Performance Models, Madison, Wisconsin*, August 1987.

[Gor81] Michael Gordon. Register Transfer Systems and Their Behavior. In *Proc. of the 5th International Conference on Computer Hardware Description Languages*, 1981.

[GSS87] Ganesh C. Gopalakrishnan, Mandayam K. Srivas, and David R. Smith. From Algebraic Specifications to Correct VLSI Circuits. In D.Borrione, editor, *From HDL Descriptions to Guaranted Correct Circuit Designs*, pages 197–225, North-Holland, 1987. (Proc of the IFIP WG 10.2 Working Conference with the same title.).

[Hen80] Peter Henderson. *Functional Programming*. Prentice Hall, 1980.

[Hen84] Matthew Hennessy. *Proving Systolic Systems Correct*. Technical Report CSR-162-84, Department of Computer Science, University of Edinburg, June 1984.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985. Definitive discussion of CSP, circa 1985.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[Hun87] Warren A. Hunt Jr. The Mechanical Verification of a Microprocessor Design. In D. Borrione, editor, *From HDL Descriptions to Guaranted Correct Circuit Designs*, Elsevier Science Publishers B.V. (North Holland), 1987. (Proc of the IFIP WG 10.2 Working Conference with the same title.).

[ISD88] I.S.Dhingra. Formal Verification of a Design Style. In Graham Birtwistle and P.A.Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 293–321, Kluwer Academic Publishers, Boston, 1988. ISBN-0-89838-246-7.

[Joh84] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. The MIT Press, 1984. An ACM Distinguished Dissertation-1983.

[Kah74] Gilles Kahn. The Semantics of a Simple language for Parallel Programming. In *IFIP-74*, North-Holland, 1974.

[Kar84] Kevin Karplus. *A Formal Model for MOS Clocking Disciplines*. Technical Report 84-632, Cornell University, Dept. of Computer Science, Cornell Univ., Ithaca, NY, 1984.

[Lin85] Gary Lindstrom. Functional Programming and the Logical Variable. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 266–280, January 1985.

[LS75] Barbara Liskov and S.N.Zilles. Specification Techniques for Data Abstractions. *IEEE Transactions on Software Engineering*, SE-1(1):7–19, 1975.

[MH85] M.Lam and H.T.Kung. A Transformational Approach to Systolic System Design. *IEEE Computer*, 18(2), 1985.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980. LNCS 92.

[Mil82] Robin Milner. *Calculii for Synchrony and Asynchrony*. Technical Report CSR-104-82, Univ. of Edinburg, 1982. Internal Report.

[Mil83] George J. Milne. CIRCAL: A calculus for circuit description. *Integration*, (1):121–160, 1983.

[Mil85a] George J. Milne. CIRCAL and the Representation of Communication, Concurrency, and Time. *ACM Transactions on Programming Languages and Systems*, 7(2):270–298, April 1985.

[Mil85b] George J. Milne. Simulation and Verification: Related Techniques for Hardware Analysis. In *Proceedings of the Seventh International Conference on Computer Hardware Description Languages*, pages 404–417, North-Holland, 1985.

[Mos83] Benjamin C. Moszkowski. *Reasoning about Digital Circuits*. PhD thesis, Stanford University, July 1983. Technical Report.

[Mue87] Eric G. Muehle. *FROBS: A Merger of Two Knowledge Representation Paradigms*. Master's thesis, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, December 1987.

[Noi82] David Noice. A Clocking Discipline for Two-Phase Digital Systems. In *Proc. International Conference on Circuits and Computers*, pages 108–111, 1982.

[Pat85] Dorab Patel. nuFP: An Environment for the Multi-level Specification, Analysis and Synthesis of Hardware Algorithms. In *Proceedings of the Functional Programming and Computer Architecture Conference*, Springer-Verlag, LNCS 201, September 1985. Nancy, France.

[She84] Mary Sheeran. muFP, a Language for VLSI Design. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 104–112, 1984.

[She85] Mary Sheeran. Design of Regular Hardware Structures Using Higher Order Functions. In *Proceedings of the Functional Programming and Computer Architecture Conference*, Springer-Verlag, LNCS 201, September 1985. Nancy, France.

[Tan87] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice Hall, Englewood Cliffs, NJ, 1987. ISBN 0-13-637406-9.

[Wei87] 1987. (Personal Communication with the Chief Architect of NS-32032.).